INITIATION À L'ALGORITHMIQUE ET À LA PROGRAMMATION EN C

Cours avec 129 exercices corrigés

Rémy Malgouyres
Professeur à l'université d'Auvergne

Rita Zrour
Maître de conférences à l'université de Poitiers

Fabien Feschet
Professeur à l'université d'Auvergne

2e édition

DUNOD

Illustration de couverture : Geometric figures-1© 25-Fotolia.com

ISBN 978-2-10-055903-9

Table des matières

PARTIE 4

STRUCTURES DE DONNÉES

Annexes

AVANT-PROPOS

QUE CONTIENT CE LIVRE ?

Cet ouvrage est destiné aux étudiants de première année des filières informatique (L1, DUT, certaines licences professionnelles), et à tous ceux qui veulent acquérir des bases solides en programmation, sans connaissances préalables, avec la volonté d'approfondir. Il inclut la programmation en langage *C* (syntaxe, exécution conditionnelle, boucles itératives, tableaux, fichiers, allocation dynamique de mémoire, récursivité...), les algorithmes et complexité (langage algorithmique, complexité d'algorithmes, tris...), et les structures de données (listes chaînées, piles, files, arbres, graphes et parcours de graphes). L'un des objectifs fixés lors de la rédaction de ce livre était de produire un ouvrage digeste. Le texte ne vise pas l'exhaustivité sur le langage *C*.

À L'ÉTUDIANT

Ce livre permet d'aborder la programmation en langage *C* sans connaissance préalable de l'informatique. Il est conçu pour pouvoir être utilisé comme un outil d'apprentissage en autodidacte. L'étudiant peut utiliser ce livre, et notamment les exercices corrigés, en complément de l'enseignement qu'il reçoit par ailleurs. L'étudiant peut aussi apprendre seul, en abordant les chapitres dans l'ordre, en contrôlant ses connaissances avec les exercices corrigés et avec les travaux pratiques sur machine.

Le texte présente de manière concise les bases qui sont nécessaires aux exercices. Les exercices de chaque chapitre sont progressifs et doivent de préférence être traités dans l'ordre. Une indication de la difficulté des exercices est donnée par des étoiles lors de l'énoncé.



Les pièges et les erreurs les plus courants sont mis en évidence clairement dans le texte par des panneaux spéciaux *Attention!*



Compléments

√ Des compléments sont donnés au fil du texte ; ils apportent un éclairage sur certaines notions difficiles et peuvent être abordés lors d'une seconde lecture ou lors de la phase d'exercices.

Des annexes expliquent comment compiler un programme *C* sur son ordinateur personnel avec le compilateur gratuit `gcc`. Des sujets supplémentaires de travaux pratiques sur machine pour chaque chapitre sont disponibles sur le site web de l'auteur Rémy Malgouyres http://www.malgouyres.fr/.

À l'enseignant

Le langage *C*, dont découlent de très nombreux autres langages actuels, reste la référence pour la programmation bas niveau. En particulier, ce langage est très présent en programmation système et réseaux. L'apprentissage de l'algorithmique en *C* permet notamment de percevoir la gestion mémoire à partir d'une gestion manuelle, qui apporte une meilleure compréhension des constructeurs et destructeurs du *C++* ou des subtilités du *garbage collector* en *Java*... L'apprentissage des structures de données en *C* permet de comprendre en détail la nature des objets manipulés à travers les librairies *Java* ou la *STL* du *C++*. L'abord de la programmation par le *C* est le moyen le plus efficace de former des informaticiens complets, ayant au final une maîtrise parfaite du bas niveau comme du haut niveau et une compréhension profonde des concepts.

L'enseignant pourra s'appuyer sur la structure de l'ouvrage, qui permet de dérouler le contenu sans référence à ce qui suit. L'exposé, mais surtout les exercices contiennent de très nombreux exemples qui peuvent être réutilisés. L'ouvrage est conçu pour un apprentissage autonome. Les exercices de chaque chapitre sont progressifs. En complément des exercices propres à chaque enseignant, les exercices du livre peuvent être donnés à faire aux étudiants qui peuvent consulter la solution *a posteriori*.

Partie 1

Bases du langage *C*

Qu'est-ce qu'un ordinateur ?

1

1.1 Exemples d'applications de l'informatique

Voici quelques exemples d'utilisation des ordinateurs :

- **Bureautique** L'ordinateur emmagasine des données saisies par une secrétaire ou autre (textes, chiffres, fichiers clients, etc.) ou des données issues d'archives, et les met en forme pour permettre une compréhension synthétique, un affichage, ou une communication de ces données.
- **Jeux vidéo** L'ordinateur combine des données entrées par le concepteur du jeu (données sur l'univers) avec les événements créés par l'utilisateur du jeu (clics de souris, etc...) pour générer des images, du son, etc.
- **Prévision météorologique** À partir de la donnée des relevés de toutes les stations météo d'une zone géographique, l'ordinateur calcule une situation future et génère des cartes de températures et de pressions atmosphériques.
- **Applications multimédia sur Internet** L'ordinateur *télécharge* des données stockées sur un *serveur* distant et affiche ces données sur l'ordinateur de l'utilisateur. Éventuellement, des actions de l'utilisateur peuvent influer sur les données affichées (on parle alors d'applications interactives).

Dans tous ces exemples, l'ordinateur *traite des données*, et produit un résultat, soit communiqué à l'utilisateur (son, images, texte), soit affiché sur un écran, ou stocké sur un disque, ou autre.

1.2 Codage des données

Les données informatiques sont toujours, en fin de compte, **codées en binaire**, c'est-à-dire qu'elles sont représentées par des suites de 0 et de 1. En effet, les données binaires sont plus faciles à mémoriser sur des supports physiques (bandes magnétiques, disques, etc.).

Par exemple, si l'on veut stocker un nombre entier sur le disque dur d'un ordinateur, on code généralement ce nombre en base 2 au lieu de le coder en base 10 comme nous y sommes naturellement habitués.

Ainsi le nombre 12 (en base 10) sera codé en base 2 par la suite binaire 00001100, ce qui signifie que :

$$\begin{aligned} 12 &= 0+0+0+0+8+4+0+0 \\ &= \mathbf{0}\times 2^7+\mathbf{0}\times 2^6+\mathbf{0}\times 2^5+\mathbf{0}\times 2^4+\mathbf{1}\times 2^3+\mathbf{1}\times 2^2+\mathbf{0}\times 2^1+\mathbf{0}\times 2^0 \end{aligned}$$

Une donnée égale soit à 0 soit à 1 s'appelle un *bit*. Une séquence de 8 bits consécutifs s'appelle un *octet* (en anglais *byte*). On mesure la quantité de mémoire stockée dans les ordinateurs en :

- Octets : 1 octet = 8 bits ;
- Kilo-octets (en abrégé Ko ou en anglais Kb) : un Ko vaut 1024 octets.
- Méga-octets (en abrégé Mo ou Mb) : un Mo vaut 1 048 576 octets
- Giga-octets (en abrégé Go ou Gb) : un Go vaut 1 073 741 824 octets

L'apparition des nombres 1024, 1 048 576 et 1 073 741 824 peut paraître surprenante, mais ce sont des puissances de 2. On retient en général qu'un Ko fait environ mille octets, un Mo environ un million, et un Go environ un milliard.

1.3 Fonctionnement d'un ordinateur

1.3.1 Système d'exploitation

Un programme informatique doit recevoir des données pour les traiter, et produire d'autres données. Pour que le programme puisse fonctionner, il faut du matériel (composants électroniques), et il faut une couche logicielle intermédiaire avec le matériel, appelée *système d'exploitation*. Le système assure la communication entre le programme informatique et le matériel, et permet au programme d'agir sur le matériel.

1.3.2 Processeur

Le processeur effectue des opérations (par exemple des opérations arithmétiques comme des additions ou des multiplications). Ces opérations sont *câblées* dans le processeur, c'est-à-dire qu'elles sont effectuées par des circuits électroniques pour être efficaces. Avec le temps, de plus en plus d'opérations complexes sont câblées au niveau du processeur, ce qui augmente l'efficacité. La vitesse d'un processeur, c'est-à-dire en gros le nombre d'opérations par seconde, appelée *vitesse d'horloge*, est mesurée en hertz (Hz), kilohertz ($1kHz = 1000Hz$) , megahertz ($1MHz = 10^6Hz$, et gigahertz ($1GHz = 10^9Hz$). Sur les architectures récentes, la puce contient plusieurs *cores*, chaque core étant l'équivalent d'un processeur et les cores communiquant entre eux très rapidement par des *bus* de données. Pour la personne qui programme en C, la configuration et la structure de la puce est *transparente*, c'est-à-dire que l'on n'a pas à s'en préoccuper (sauf pour l'optimisation en programmation très avancée).

1.3.3 Mémoire centrale

Au cours du déroulement du programme, celui-ci utilise des données, soit les données fournies en entrée, soit des données intermédiaires que le programme utilise pour fonctionner. Ces données sont stockées dans des *variables*. Physiquement, les variables sont des données binaires dans la *mémoire centrale* (appelée aussi mémoire *RAM*). La mémoire centrale communique rapidement avec le processeur. Lorsque le processeur effectue un calcul, le programmeur peut indiquer que le résultat de ce calcul doit être mémorisé dans une variable (en *RAM*). Le processeur pourra accéder plus tard au contenu de cette variable pour effectuer d'autres calculs ou produire un résultat en sortie. La quantité de mémoire *RAM* est mesurée en octets (ou en mégaoctets ou gigaoctets). Les données en mémoire centrale ne sont conservées que pendant le déroulement du programme, et disparaissent lorsque le programme se termine (notamment lorsque l'on éteint l'ordinateur).

1.3.4 Périphériques

Le programme reçoit des données des périphériques en entrée, et communique ses résultats en sortie à des périphériques. Une liste (non exhaustive) de périphériques usuels est :

- le clavier qui permet à l'utilisateur de saisir du texte ;
- la souris qui permet à l'utilisateur de sélectionner, d'activer ou de créer à la main des objets graphiques ;
- l'écran qui permet aux programmes d'afficher des données sous forme graphique ;
- l'imprimante qui permet de sortir des données sur support papier ;
- le disque dur ou la clef *USB* qui permettent de stocker des données de manière permanente. Les données sauvegardées sur un tel disque sont préservées, y compris après terminaison du programme ou lorsque l'ordinateur est éteint, contrairement aux données stockées en mémoire centrale qui disparaissent lorsque le programme se termine.

Les périphériques d'entrée (tels que le clavier et la souris) transmettent les données dans un seul sens, du périphérique vers la mémoire centrale. Les périphériques de sortie (tels que l'écran ou l'imprimante) recoivent des données dans un seul sens, de la mémoire centrale (ou de la mémoire vidéo) vers le périphérique. Les périphériques d'entrée-sortie (tels que le disque dur, le port USB, ou la carte réseau) permettent la communication dans les deux sens entre la mémoire centrale et le périphérique.

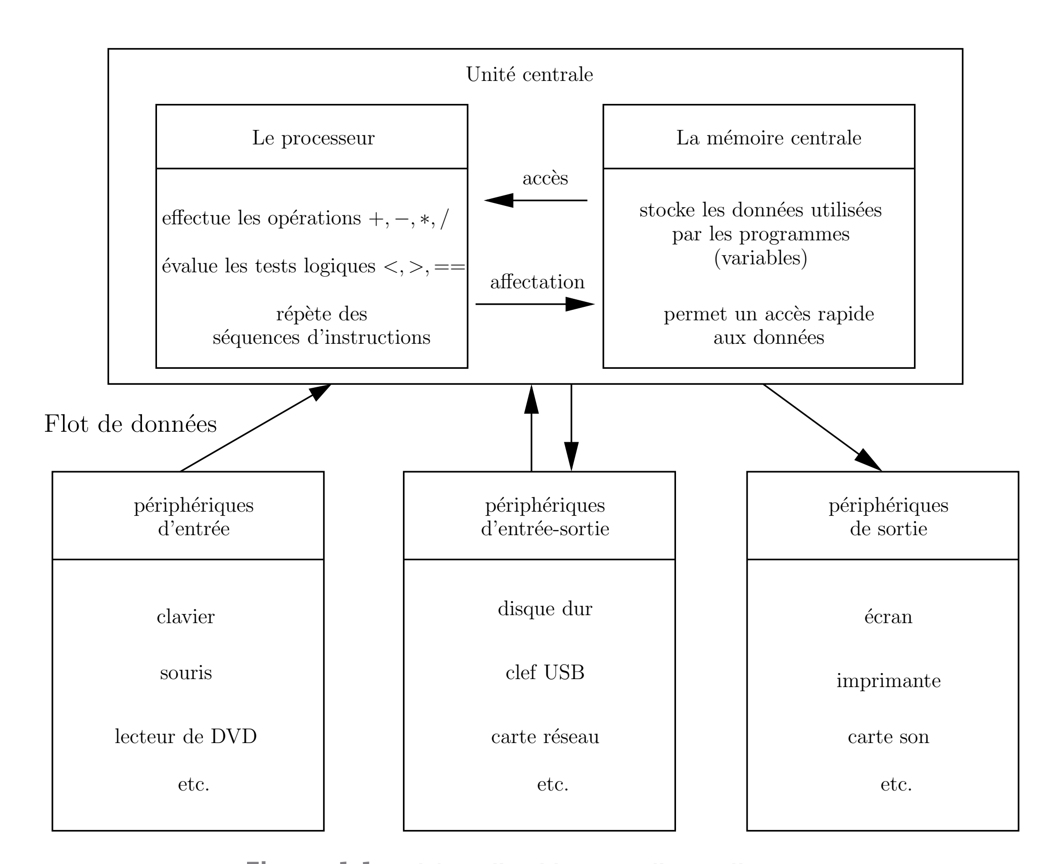


Figure 1.1 – Schéma d'architecture d'un ordinateur.

PREMIERS PROGRAMMES

2

2.1 QU'EST-CE QU'UN PROGRAMME ?

Un programme informatique réalise en général trois choses :

- **Il lit des données en entrée**. Le programme doit en effet savoir à partir de quoi travailler. Par exemple, pour utiliser une calculatrice, on doit lui donner des nombres et lui dire quelles opérations effectuer. Pour cela, on utilise souvent un *clavier*, mais le programme peut aussi tirer les données d'un *disque dur* ou encore d'un *autre ordinateur via* un réseau ou autre.
- **Il effectue des calculs**. À partir des données en entrée, le programme va appliquer automatiquement des méthodes pour traiter ces données et produire un résultat. Les méthodes que sont capables d'effectuer les ordinateurs s'appellent des *algorithmes*. Par exemple, une calculatrice va appliquer l'algorithme d'addition ou de multiplication.
- **Il écrit des données en sortie**. Lorsque le programme a obtenu un résultat, il doit écrire ce résultat quelque part pour qu'on puisse l'utiliser. Par exemple, une calculatrice va afficher un résultat à l'*écran* ou stocker le résultat en *mémoire*.

Le travail d'un programmeur consiste à créer des programmes informatiques. Le programmeur doit pour cela expliquer à l'ordinateur dans un certain langage, appelé *langage de programmation*, quelles sont les données et quelles sont les méthodes à appliquer pour traiter ces données. Dans ce chapitre, nous verrons en langage *C*, les premiers exemples permettant :

1. de lire une donnée au clavier avec la fonction `scanf` ;
2. d'effectuer les calculs les plus simples sur des nombres et de stocker le résultat dans une variable ;
3. d'afficher un texte ou un nombre à l'écran avec la fonction `printf`.

Ce faisant, nous verrons la structure d'un programme *C* très simple et quelques notions sur la *syntaxe* du langage. Les notions vues dans ces exemples seront développées dans les chapitres suivants. Une fois que le programmeur a écrit son programme, qui est du texte en langage *C*, il doit *compiler* le programme pour créer un fichier *exécutable* pour qu'un utilisateur du programme puisse utiliser ce programme. Le processus de compilation est décrit en annexe.

2.2 Afficher un mot

Voici un programme *C* qui écrit un message de bienvenue : le mot “Bonjour”.

```
#include <stdio.h>          /* pour pouvoir lire et écrire */

int main(void)              /* programme principal */
{
   printf("Bonjour !\n");      /* écriture à l'écran */
   return 0;
}
```

Les phrases comprises entre /∗ et ∗/ sont des *commentaires*. Elles n'ont pas d'influence sur le déroulement du programme. Les (bons) programmeurs mettent des commentaires pour clarifier leur code, ce qui est crucial lorsqu'on travaille en équipe.

La première ligne est une instruction `#include` < `stdio.h` > qui permet d'utiliser les fonctions de la bibliothèque `stdio.h` dans le programme. Cette bibliothèque contient notamment les fonctions d'affichage à l'écran `printf` et de lecture au clavier `scanf`.

Vient ensuite la ligne déclarant le début de la fonction `main`, le programme principal. Le programme principal est la suite des instructions qui seront exécutées. En l'occurrence, il n'y a qu'une seule instruction : un affichage à l'écran par `printf`. Le `\n` permet de passer à la ligne après l'affichage du mot “Bonjour”.

2.3 Lire un nombre

Voici un programme permettant à l'utilisateur de taper un nombre au clavier. Ce nombre est lu par le programme et mémorisé dans une variable `x` qui est un nombre réel (type de données `float`). La variable `x` est ensuite ré-affichée par `printf`.

```
#include <stdio.h>          /* pour pouvoir lire et écrire */

int main(void)              /* programme principal */
{
   float x; /* déclaration d'une variable x (nombre réel) */

   printf("Veuillez entrer un nombre réel au clavier\n");
   scanf("%f", &x); /* lecture au clavier de la valeur de x */
   /* affichage de x : */
   printf("Vous avez tapé %f, félicitations !", x);
   return 0;
}
```

Le format d'affichage et de lecture `%f` correspond à des nombres réels (type `float`). Dans `printf`, lors de l'affichage, le `%f` est remplacé par la valeur de *x*. Par exemple, si l'utilsateur a entré la valeur `15.6` au clavier, le programme affiche la phrase suivante :

```
Vous avez tapé 15.600000, félicitations !
```



Ne pas oubier le & dans le `scanf` ! Cela provoquerait une erreur mémoire (ou *erreur de segmentation*) lors de l'exécution du programme, et le programme serait brutalement interrompu.

2.4 Effectuer un calcul et mémoriser le résultat

Le programme suivant mémorise le double de x dans une variable y, par le biais d'une *affectation*. L'affectation (symbole =) permet de stocker le résultat d'un calcul dans une variable.

```
#include <stdio.h>          /* pour pouvoir lire et écrire */

int main(void)              /* programme principal */
{
   float x, y;   /* déclaration de deux variables x et y */

   printf("Veuillez entrer un nombre réel au clavier\n");
   scanf("%f", &x); /* lecture au clavier de la valeur de x */
   y = 2*x; /* on met dans y le double du contenu de x */
   printf("Le double du nombre tapé vaut %f \n", y);
   return 0;
}
```



Le symbole = de l'affectation a une toute autre signification que l'égalité mathématique. L'affectation signifie qu'une variable prend la valeur du résultat d'un calcul. Il correspond à une opération de recopie d'une donnée.



Compléments

√ La syntaxe doit être respectée rigoureusement. Si on oublie un point-virgule, ou bien si on remplace par exemple un guillemet " par une quote ', cela provoque en général une erreur à la compilation. Dans certains cas très précis, il y a plusieurs possibilités pour la syntaxe. Par exemple, le mot `void` dans la déclaration du `main` est facultatif.

√ Parfois, surtout en phase de conception, un programme peut être syntaxiquement correct, mais le comportement du programme n'est pas celui souhaité

par le programmeur, en raison d'une erreur de conception ou d'inattention. On dit que le programme comporte un *bogue* (en anglais *bug*).

√ Le `return 0` à la fin du `main` indique seulement que la valeur 0 est retournée au système (ce qui indique que le programme ce termine sans erreur). Nous n'utiliserons pas la possibilité de retourner des valeurs au système. Ces fonctionalités sont généralement étudiées avec les systèmes d'exploitation.

Exercices

2.1 (∗) Pour convertir des degrés Fahrenheit en degrés Celsius, on a la formule suivante :

$$C \simeq 0.55556 \times (F - 32)$$

où F est une température en degrés Fahrenheit et C la température correspondante en degrés Celsius.

a) Écrire un programme *C* qui convertit une température entrée au clavier exprimée en degrés Fahrenheit et affiche une valeur approchée de la même température en degrés Celsius. Les températures seront exprimées par des nombres réels.

b) Même question qu'au a) pour la conversion inverse : de degrés Celsius en degrés Fahrenheit.

2.2 (∗) Lors d'une opération de promotion, un magasin de composants *hardware* applique une réduction de 10% sur tous les composants. Écrire un programme qui lit le prix d'un composant au clavier et affiche le prix calculé en tenant compte de la réduction.

2.3 (∗∗) Soit la fonction mathématique f définie par $f(x) = (2x + 3)(3x^2 + 2)$

a) Écrire un programme *C* qui calcule l'image par f d'un nombre saisi au clavier.

b) Une approximation de la dérivée f' de la fonction f est donnée en chaque point x, pour h assez petit (proche de 0), par :

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h}.$$

Écrire un programme *C* qui calcule et affiche une approximation de la dérivée de f en un point x entré au clavier. On pourra faire saisir le paramètre h au clavier.

2.4 (*c*-∗) Une bille de plomb est lâchée du haut d'un immeuble et tombe en chute libre. Au bout d'un temps t (exprimé en secondes), la bille est descendue d'une hauteur (en mètres) :

$$h = \frac{1}{2}g.t^2$$

avec

$$g = 9.81 \text{ (exprimé en } (m.s^{-2}))$$

a) Écrire un programme qui calcule la hauteur descendue au bout d'un temps t saisi au clavier.

b) Écrire un programme qui calcule la durée totale de la chute connaissant la hauteur totale h de l'immeuble saisi au clavier. (On pourra utiliser la fonction `sqrt` de la bibliothèque `math.h` qui calcule la racine carrée d'un nombre.)

Corrigés

2.1

a)

```
int main(void)
{
  float celsius, fahrenheit;
  printf("Entrez une température en degrés Fahrenheit : ");
  scanf("%f", &fahrenheit);
  celsius = 0.55556 * (fahrenheit - 32.0);
  printf("Température de %f degré Celsius.\n", celsius);
  return 0;
}
```

b)

```
int main(void)
{
  float celsius, fahrenheit;
  printf("Entrez une température en degrés Celsius : ");
  scanf("%f", &celsius);
  fahrenheit = (celsius / 0.55556) + 32.0;
  printf("Température de %f degré Fahrenheit.\n", fahrenheit);
  return 0;
}
```

2.2

```
int main(void)
{
  float prix, prixRemise;
  printf("Entrez un prix : ");
  scanf("%f", &prix);
  prixRemise = 0.9 * prix;
  printf("Le prix avec 10 %% de remise est de %f.\n", prixRemise);
  return 0;
}
```

2.3

a)

```
int main(void)
{
  float x, fx;
  printf("Entrez un nombre : ");
  scanf("%f", &x);
  fx = (2.0 * x + 3.0) / (3.0 * x * x + 2.0);
  printf("f(%f) = %f\n", x, fx);
  return 0;
}
```

b)

```
int main(void)
{
  float x, h, fx, fx_plus_h, fPrime_x;
  printf("Entrez un nombre : ");
  scanf("%f", &x);
  printf("Entrez un écart h : ");
  scanf("%f", &h);
  fx = (2.0 * x + 3.0) / (3.0 * x * x + 2.0);
  fx_plus_h = (2.0 * (x + h) + 3.0) / (3.0 * (x + h) * (x + h) + 2.0);
  fPrime_x = (fx_plus_h - fx) / h;
  printf("f'(%f) = %f\n", x, fPrime_x);
  return 0;
}
```

2.4

a)

```
int main(void)
{
  float h, t;
```

```
  printf("Entrez une durée : ");
  scanf("%f", &t);
  h = (9.81 * t * t) / 2.0;
  printf("A t = %f, h = %f\n", t, h);
  return 0;
}
```

b) Ne pas oubliez d'ajouter `-lm` à la compilation

```
int main(void)
{
  float h, t;
  printf("Entrez la hauteur totale (en mètres) : ");
  scanf("%f", &h);
  t = sqrt(2.0 * h / 9.81);
  printf("La bille touche le sol au bout de %f secondes\n", t);
  return 0;
}
```

Types de données 3

3.1 Variables et opérations

Dans un programme, il apparaît des *variables* qui permettent de donner des noms à des données. Chaque variable doit avoir un *type* (nombre entier, nombre réel, caractère, suite de caractères, ou type plus complexe). Chaque variable a aussi un *identificateur* qui est le nom de la variable. Une déclaration de variable a toujours la forme :

```
type identificateur;
```

ou bien la forme avec l'initialisation (c'est-à-dire que l'on donne à la variable une valeur initiale) :

```
type identificateur = valeur;
```

On peut aussi déclarer plusieurs variables d'un même type séparées par des virgules.

> ***Exemple***
>
> ```
> float x, y=2.0; /* nombres réels. y a pour valeur initiale 2.0 */
> int n;   /* nombre entier */
> ```

Suivant le type des variables, certaines opérations sont définies sur ces variables (par exemple la multiplication dans le cas des nombres entiers, etc.).

3.2 Type entier `int`

Le type `int` (abréviation de l'anglais *integer*) est une représentation des nombres entiers. Comme toute variable informatique, un `int` ne peut prendre qu'un nombre fini de valeur. Un `int` est généralement codé sur 4 octets (32 bits). Dans ce cas, les valeurs sont entre -2^{31} et $2^{31}-1$ (ce qui fait bien 2^{32} valeurs possibles). Notons que sur certains vieux systèmes, les `int` sont codés seulement sur 2 octets.

Les opérations arithmétiques binaires $+, -, *, /$ sont définies sur les `int` et donnent **toujours pour résultat un** `int`.



Si le résultat, par exemple, d'une multiplication dépasse la limite $2^{31}-1$ de capacité d'un `int`, le résultat est tronqué, donnant un résultat parfois inattendu. On parle de *dépassement de capacité* ou en anglais *overflow*.



La division de deux entiers est une **division euclidienne** dont le résultat est toujours un entier. Par exemple, 7/2 est égal à 3 et non pas 3.5. Ceci est source de nombreux bugs. Par exemple, il arrive que le résultat d'une division entre deux variables de type `int` donne toujours 0 de manière inattendue, ce qui est dû à une division entre deux entiers dont le quotient réel est entre 0 et 1.

Le signe − peut aussi désigner l'opposé d'un nombre. On parle alors d'opération unaire et non pas binaire.

Il y a aussi une opération appelée *modulo*. Le modulo correspond au reste de la division entre deux entiers, et est noté %.

Par exemple, quand on dit

- "dans 17 combien de fois 5 ?"
- "3 fois et il reste 2"

En mathématiques, on écrit : $17 = 3 \times 5 + 2$.

En informatique, on écrit que 17/5 est égal à 3 et que 17%5 est égal à 2.

Plus généralement, on a toujours l'expression dite de la *division euclidienne* :

$$n = p * \underbrace{(n/p)}_{quotient} + \underbrace{(n\%p)}_{reste}$$

Voici un autre exemple : un programme qui affiche le chiffre des unités d'un nombre saisi au clavier.

```
#include <stdio.h>

int main(void){
   int nombre, chiffre;
   puts("Tapez un nombre entier :");
   scanf("%d", &nombre);
   chiffre = nombre%10;
   printf("Le dernier chiffre est : %d", chiffre);
   return 0;
}
```

3.3 Les types réels float et double

Les types `float` et `double` permettent de représenter des nombres réels avec une certaine précision (suivant une représentation des nombres appelée *virgule flottante* ou nombre flottant).

Un nombre réel tel qu'il est ainsi représenté possède une *mantisse* (des chiffres) et un *exposant* qui correspond à la multiplication par une certaine puissance de 10. Par exemple $3.546E - 3$ est égal à 3.546×10^{-3}, ce qui fait 0.003546.

Le type `double` (codé sur 8 octets) est plus précis que le type `float` (codé sur 4 octets). La valeur maximale d'un `double` est d'environ 10^{308} alors que celle d'un `float` est de l'ordre de 10^{38}. (Les valeurs limites exactes sont données par les constantes `DBL_MAX` et `FLT_MAX` de la bibliothèque `float.h`).

Sur les nombres réels sont définies les quatres opérations binaires $+, -, *, /$, ainsi que le $-$ unaire.

La bibliothèque `math.h` contient les fonctions de calcul scientifique `pow` (puissance), `sqrt` (racine carrée), `cos`, `sin`, `tan`, etc.

3.4 Le type `char`

Le type `char` (abréviation de l'anglais *character*) est un type caractère codé sur 1 octet. C'est la plus petite donnée qui puisse être stockée dans une variable. Les valeurs (de -126 à 125) peuvent réprésenter des caractères conventionnels.

Par exemple, les caractères alphabétiques majuscules $A, B, \ldots, Z$ ont pour valeur $65, 66, \ldots, 90$ et les minuscules correspondantes $a, b, \ldots, z$ ont pour valeurs $97, 98, \ldots, 122$. On appelle ce codage des caractères le *code ASCII*.

Une variable de type `char` peut être considérée soit comme un nombre, soit comme un caractère que l'on peut afficher. Dans tous les cas, il s'agit du même type. Pour désigner par exemple le caractère Z dans un programme, on peut soit écrire `'Z'` (entre quotes), soit écrire 90 qui est le code *ASCII* du caractère Z. Dans les deux cas il s'agit du même caractère et de la même donnée qui peut être stockée dans la même variable de type `char`.

3.5 Les types `unsigned`

Aux types `int` et `char` correspondent des types `unsigned int` (aussi sur 4 octets) et `unsigned char` (aussi sur 1 octet) qui représentent uniquement des valeurs positives.

Un `unsigned int` sur 4 octets va de 0 à $2^{32} - 1$.

Un `unsigned char` sur 1 octet va de 0 à $2^8 - 1$ (c'est-à-dire de 0 à 255).

3.6 Affectations et conversions

Étant données deux variables de même type, on peut recopier le contenu d'une variable dans l'autre par une *affectation* (signe =). Plus généralement, on peut copier dans une variable la valeur de toute une expression.

Si les deux variables sont de types différents, l'opération d'affectation = va réaliser, lorsque c'est possible, une conversion. Si la conversion n'est pas possible, le compilateur affichera un message d'erreur.

Exemple d'affectation

```
int var1=0, var2=3; /* déclarations avec initialisations */
var1=2*var2+1; /* affectation : après ceci var1 vaut 7 */
```

***Exemple de conversion d'un* `float` *vers un* `int`**

```
int n;
float x=7.6587e2;
n = (int)x;   /* après ceci n vaut 765 (arrondi à la partie entière) */
```

***Exemple de conversion d'un* `unsigned int` *vers un* `unsigned char`**

```
unsigned char c;
unsigned int n = 321;
c = (unsigned char)n;  /* après ceci c vaut 65 (321 modulo 256) */
                       /* c'est-à-dire que c vaut 'A'  */
```

Lorsqu'on affecte (par exemple) un `double x` à un `int n`, il y a en général une perte d'information. Le compilateur nous donne un message d'avertissement (*warning*), à moins que l'on effectue un *cast*, appelé aussi *conversion explicite*, en indiquant le type `int` entre parenthèses :

```
n = (int)x;  /* Affectation avec conversion explicite (cast) */
```

Un dernier exemple : calculer la valeur précise d'un quotient ($\frac{2}{3}$ sur l'exemple suivant) de deux entiers :

```
int n=2, m=3;
double deuxtiers;
/* conversion avant division : */
deuxtiers = ((double)n)/((double) m);
```

Une autre possibilité est d'écrire :

```
double deuxtiers;
deuxtiers = 2.0/3.0;
```



Pour le calcul de `deuxtiers` ci-dessus, écrire `deuxtiers=n/m` ou encore `deuxtiers=2/3` est une grave erreur qui conduit au résultat que `deuxtiers` est égal à 0, car le quotient entre entiers donne une division euclidienne, même lorsqu'on l'affecte ensuite à un `float` ou un `double`.

3.7 Les constantes et le `#define`

Une *constante* est une valeur qui n'est pas susceptible de varier lors de l'exécution d'un programme. Par exemple, 9.81 est une constante de type `float`, 1024 est une constante de type `int` (qui peut aussi être affectée à une variable de type `float`).

Enfin, 65, ou 'A' est une constante de type `char` (qui peut aussi être affectée à un `int` ou un `float`).

On peut donner un nom à une constante (ce qui évite de répéter des chiffres au risque de se tromper et augmente la souplesse du programme) par un `#define` au début du programme (après les `#include`) :

```
#define PI  3.14159265358979323846  /* Constante de type float */
#define G 9.81      /* Constante de type float */
#define H 4816      /* Constante de type int */

/* Voici aussi une constante de type chaîne de caractères :  */
#define MESSAGE1 "Erreur, vous devez mettre le #define hors du main !"
```

Le `#define` n'est pas une instruction comme les autres. Comme le `#include`, c'est une *directive de précompilation*.



Ne pas confondre les constantes de type caractère (`char`), avec des simples quotes comme `'Z'`, et les constantes de types chaînes de caractères, entre double quotes, comme `"Voici une phrase"` qui peuvent contenir plusieurs caractères. La chaîne de caractère `"Z"` existe aussi, c'est un cas particulier d'une chaîne avec une seule lettre, mais elle n'est pas de type `char` (nous étudierons plus loin le type chaîne).



Ne pas confondre le caractère `'6'`, dont le code *ASCII* vaut 54, avec le caractère 6, dont le code *ASCII* vaut 6, mais qui représente un autre caractère.

3.8 Définir ses propres types

En *C*, le programmeur peut définir ses propres types et donner les noms qu'il veut à ses types. Pour donner un nom à un type, on utilise `typedef`. Dans l'exemple suivant, le programmeur a préféré appeler les nombres entiers `Entier` plutôt que `int`.

```
#include <stdio.h>

typedef int Entier; /* Définition d'un nouveau type Entier */

int main(void)
{
   Entier d, n;
   scanf("%d", &n);
   d = 2*n;
   printf("Le double de %d est %d\n", n, d);
   return 0;
}
```

Dans ce cas, le type `Entier` est simplement sinonyme de `int`. Nous verrons plus loin comment définir des types plus compliqués pour représenter des informations de toutes sortes. Les différents types définis et utilisés dans un programme s'appellent les *structures de données*.

Exercices

3.1 (∗) Sachant que le premier avril 2004 était un jeudi, écrire un programme qui détermine le jour de la semaine correspondant au 4 mai de la même année. On pourra représenter les jours de la semaine par des numéros allant de 0 à 6.

3.2 (∗) Écrire un programme qui lit un nombre au clavier, répond 1 si le nombre est impair et 0 si le nombre est pair.

3.3 (∗∗) Écrire un programme qui affiche le chiffre des dizaines d'un nombre saisi au clavier. Même question pour les centaines.

3.4 (∗) Écrire un programme qui arrondi un nombre réel saisi au clavier à deux chiffres après la virgule.

3.5 (∗) Écrire un programme qui lit un nombre *r* au clavier et calcule le périmètre et l'aire d'un disque de rayon *r*.

Corrigés

3.1

```
int main(void)
{
  int njours;
  /* Convention : 0 <-> lundi, ... 6 <-> Dimanche
     33 jours écoulés entre le 1er avril et le 4 mai */
  njours = (33 + 3) % 7 + 1;
  printf("Le 4 mai était le %d ème jour de la semaine.\n", njours);
  return 0;
}
```

3.2

```
int main(void)
{
  int n, parite;
  printf("Entrez un nombre : ");
```

```
  scanf("%d", &n);
  parite = n % 2;
  printf("La parité du nombre est %d\n", parite);
  return 0;
}
```

3.3

```
int main(void)
{
  int n, dizaine, centaine;
  printf("Entrez un nombre : ");
  scanf("%d", &n);
  dizaine = (n / 10) % 10;
  centaine = (n / 100) % 10;
  printf("Le chiffre des dizaines est %d\n", dizaine);
  printf("Le chiffre des centaines est %d\n", centaine);
  return 0;
}
```

3.4

```
int main(void)
{
  float n, arrondi;
  int multiplie;
  printf("Entrez un nombre réel : ");
  scanf("%f", &n);
  multiplie = (int) ((n + 0.005) * 100);
  arrondi = multiplie / 100.0;
  printf("%f arrondi à deux chiffres est %.2f\n", n, arrondi);
  return 0;
}
```

3.5

```
int main(void)
{
  float rayon, perimetre, aire;
  printf("Entrez le rayon : ");
  scanf("%f", &rayon);
  perimetre = 2.0 * M_PI * rayon;
  aire = M_PI * rayon * rayon;
  printf("Périmètre = %f\nAire = %f\n", perimetre, aire);
  return 0;
}
```

ENTRÉES-SORTIES : stdio.h

4

4.1 QU'EST-CE QU'UNE BIBLIOTHÈQUE D'ENTRÉES-SORTIES ?

Une *bibliothèque* est un ensemble de fonctionalités ajoutées à un langage de programmation. Chaque bibliothèque a un thème. Par exemple (en langage *C*) :

1. la bibliothèque `math.h` contient les fonctions mathématiques de calcul numérique et des constantes comme `M_PI` pour représenter le nombre π ;
2. la bibliothèque `time.h` contient les types et fonctions permettant de gérer la durée (date et heure, temps d'exécution du programme...) ;
3. la bibliothèque `float.h` contient les limites et constantes des types `float` et `double` ;
4. etc.
5. La bibliothèque `stdio.h` contient les types et fonctions permettant de gérer les entrées-sorties (saisies au clavier, affichage de texte, mais aussi fichiers...).

L'extension `.h` est réservée aux fichiers d'en-tête, ou fichier *header*, qui servent lorsqu'on ne peut pas mettre l'ensemble du programme dans un même fichier. Grâce aux fichiers *header*, on va pouvoir utiliser, dans un code *C*, des fonctions qui sont écrites dans d'autres fichiers, éventuellement par d'autres programmeurs, ce qui est le cas pour les bibliothèques standard telles que `stdio.h`.

Le nom `stdio.h` est un abrégé de l'anglais *Standard Input Output*. Nous verrons dans ce chapitre une partie des fonctions de cette bibliothèque qui concernent la lecture de données clavier et les affichages à l'écran.

Pour utiliser les fonctionnalités de la bibliothèque `stdio.h`, il faut mettre au début du programme

```
#include <stdio.h>
```

4.2 L'AFFICHAGE DE DONNÉES SOUS FORME DE TEXTE

4.2.1 Afficher des caractères

Pour afficher un caractère on peut utiliser la fonction `putchar` :

```
putchar('A'); /* Affiche un 'A'*/
putchar(65);  /* Affiche aussi un 'A'*/
```

Il y a une manière simple d'afficher un message (ou plus généralement une chaîne de caractères, voir plus loin) par la fonction puts :

```
puts("coucou !");
```

La fonction puts va automatiquement à la ligne après l'affichage. Pour afficher un message sans aller à la ligne, on peut utiliser printf :

```
printf("coucou !");
```

4.2.2 Afficher d'autres données

Pour afficher des nombres, il faut spécifier un *format*, c'est à dire qu'il faut préciser comment le résultat doit être affiché.

Prenons l'exemple d'un caractère affiché sous deux formats différents :

```
char caract='A';
printf("%c", caract); /* affiche en tant que caractère 'A'*/
printf("%d", caract)  /* affiche en tant que nombre 65 */
```

La spécification du format, si elle est erronée, peut provoquer un résultat incompréhensible. Parfois, plusieurs formats d'affichage sont possibles pour une même donnée. Ils produisent un affichage sous des formes différentes.

1. Pour afficher des nombres entiers, le format est %d :

```
int nombre=185;
printf("%d", nombre);
```

2. Pour afficher un réel (float ou double), un format usuel est %f :

```
float x=2.0/3.0;
printf("%f", x);
```

3. Pour afficher un réel (float ou double) avec plus de précision (plus de chiffres après la virgule), le format est %lf :

```
float x=2.0/3.0;
printf("%lf", x);
```

4. Pour afficher un réel en spécifiant le nombre de chiffres après la virgule, le format est %.?f :

```
float x=2.0/3.0;
printf("%.3f", x);  /* affichage de 0.667 */
```

5. Pour afficher un réel avec puissance de 10, le format est `%e` :

```
float x=6.02e23; /* x = 6.02 × 10^23 */
printf("%e", x);  /* affichage de 6.020000e23 */
```

6. Pour afficher un caractère, le format usuel est `%c` :

```
char caract = 68;
printf("%c", caract); /* Affichage de 'D'*/
```

7. Pour afficher une chaîne de caractères, le format est `%s` :

```
printf("%s", "coucou !");
```

Avec la fonction `printf`, on peut aussi combiner l'affichage de messages avec l'affichage de données numériques :

```
double f=25.0/6.0;
int entier = (int) f;  /* conversion explicite (cast) */
printf("La partie entière de %f est %d\n", f, entier);
printf("La partie fractionnaire de %f est %f", f, f-entier);
```

Rappelons que le caractère spécial `'\n'` provoque un retour à la ligne.

4.3 Lecture au clavier

Comme dans le cas de l'affichage, des fonctions telles que `getchar` (qui permet de lire un caractère) ou autres permettent de lire des données texte. La fonction `scanf` permet de lire des données avec les formats suivants (liste non exhaustive) :

- `%d` pour le type *int* ;
- `%u` pour le type *unsigned int* ;
- `%f` pour le type *float* ;
- `%lf` pour le type *double* ;
- `%c` pour le type *char* ;
- `%s` pour le type chaîne (étudié plus loin).



Ne pas oublier le `&` devant chaque variable dans `scanf`, cela provoquerait une erreur mémoire (erreur de segmentation).



Contrairement aux erreurs sur le format d'affichage, qui provoquent en général affichage incompréhensible, un mauvais format dans `scanf` peut provoquer une erreur mémoire. Notamment, dans `scanf`, les formats `%f` et `%lf` sont incompatibles et dépendent strictement du type de données (`float` ou `double`).

Notons que l'on peut lire plusieurs valeurs dans un même appel à `scanf`, en séparant les `%d`, `%f`,... par des espaces.

Exemple

```
int nombre;
float f1;
double f2;

Puts("Tapez un nombre entier :")
scanf("%d", &nombre);
printf("Vous avez tapé %d, bravo !\nEssayons encore", nombre);
printf("\nTapez deux nombres réels\n");
scanf("%f %lf", &f1, &f2);
printf("Vous avez tapé : %f et %f. Est-ce bien cela ?\n", f1, f2);
```

Dans ce dernier `scanf`, l'utilisateur peut saisir un espace ou bien un retour chariot (touche "Entrée") entre la saisie des deux nombres réels. Si on mettait un autre caractère que l'espace dans `scanf`, comme `*` dans l'exemple suivant, cela obligerait l'utilisateur à saisir un caractère précis.

```
float f1, f2;

printf("\nTapez deux nombres réels\n");
puts("IMPERATIVEMENT séparés par une * ");
scanf("%f*%f", &f1, &f2);
printf("Vous avez tapé : %f et %f. Est-ce bien cela ?\n", f1, f2);
```



Dans l'exemple précédent, si l'utilisateur n'entre pas une `*`, le déroulement du programme reste bloqué sur le `scanf`.

Exercices

4.1 (*c*) Écrire un programme qui affiche le code ASCII d'un caractère saisi au clavier.

4.2 (∗∗) Écrire un programme qui :

1. lit deux entiers `n1` et `n2` au clavier ;

2. affiche la partie entière de leur quotient ;

3. affiche la partie fractionnaire `frac` de leur quotient ;

4. lit un nombre réel `l` au clavier ;

5. calcule la partie entière du produit de `l` par `frac` modulo 256, puis convertit le résultat en caractère ;

6. affiche le caractère obtenu.

Tester avec $n1 = 1321$, $n2 = 500$, $l = 500$.

Corrigés

4.1

```
int main(void)
{
  char c;
  printf("Tapez un caractère : ");
  scanf("%c", &c);
  printf("Le code ASCII de %c est %d\n", c, c);
  return 0;
}
```

4.2

```
int main(void)
{
  int n1, n2;
  int quotient;
  float frac, l;
  char c;
  printf("Entrez deux nombres entiers : ");
  scanf("%d %d", &n1, &n2);
  quotient = n1 / n2;
  printf("Partie entière du quotient : %d\n", quotient);
  frac = (n1 / (float) n2) - quotient;
  printf("Partie fractionnaire du quotient : %f\n", frac);
  printf("Entrez un réel : ");
  scanf("%f", &l);
  c = (char) ((int) (l * frac) % 256);
  printf("Caractère : %c\n", c);
  return 0;
}
```

5 EXÉCUTION CONDITIONNELLE

5.1 QU'EST-CE L'EXÉCUTION CONDITIONNELLE ?

L'exécution conditionnelle permet de faire deux choses différentes selon le cas qui se produit. L'instruction ne sera exécutée que sous certaines conditions. Plus précisément, le programme teste une condition, si la condition est satisfaite le programme fait une chose, dans le cas contraire, le programme fait une autre chose. Une instruction conditionnelle peut avoir plusieurs formes.

5.2 CONDITION `si-alors`

Supposons que l'on ait une `condition` (par exemple que l'âge du capitaine soit inférieur à 30 ans). Si la condition est vérifiée, on fait quelque chose, dans le cas contraire, on ne fait rien. En algorithmique, cela s'écrit :

```
série d'instructions 1  /* début du programme */
   si (condition)
      alors série d'insructions 2
   fin si
   série d'instructions 3 /* suite du programme */
```

Cela signifie que la série d'instructions 2 sera exécutée seulement si la condition est réalisée.
Voyons un exemple en langage *C*.

```
char choix;

puts("Attention, ce programme est susceptible de");
puts("heurter la sensibilité de ceux d'entre vous");
puts("qui sont allergiques aux maths !");

puts("\nVoulez-vous continuer ? (y/n)");

choix = getchar(); /* getchar permet de saisir un caractère */

if (choix == 'y')
{
   puts("Le carré de l'hypoténuse est égal à");
   puts("la somme des carrés des deux autres côtés !");
}
```

```
if (choix == 'n')
   puts("Bon, bon, tant pis pour vous...");
```

Les instructions au conditionnel doivent former un *bloc*. Un bloc peut être formé d'une seule instruction (et donc un seul point-virgule), ou bien de plusieurs instructions comprises entre des accolades `{ }`. Autrement dit, les accolades encadrant les instructions sur lesquelles porte le `if` sont facultatives **dans le cas où** il n'y a qu'une seule instruction dans le `if`.



Ne pas oublier les accolades { } lorsque le `if` porte sur plusieurs instructions, faute de quoi seule la première instruction serait au conditionnel et les instructions suivantes ne le seraient pas, ce qui provoquerait un bug.



Ne pas mélanger le signe = qui fait une affectation et le signe == qui teste l'égalité de deux expressions. Cette erreur ne serait pas détectée par le compilateur et provoquerait un bug.

5.3 Condition `si-alors-sinon`

Les instructions conditionnelles sont souvent de la forme suivante :

```
série d'instructions 1  /* début du programme */
   si (condition)
   alors
      série d'instructions 2
   sinon
      série d'instructions 3
   fin si
   série d'instructions 4 /* suite du programme */
```

En d'autres termes, dans un cas, on fait une chose, et dans le cas contraire, on fait une autre chose. En reprenant l'exemple précédent :

```
...
puts("\nVoulez-vous continuer ? (y/n)");

choix = getchar();

if (choix == 'y')
{
   puts("Le carré de l'hypoténuse est égal à");
   puts("la somme des carrés des deux autres côtés !");
}
else
   puts("Bon, bon, tant pis pour vous...");
```

Notons que dans ce cas, si l'utilisateur tape n'importe quel caractère autre que `'y'` (pas seulement `'n'`), le programme rentre dans le cas `else`. À nouveau, les accolades sont facultatives s'il n'y a qu'une seule instruction.

5.4 Notions de calcul booléen

5.4.1 Expressions de base

Les conditions que l'on peut mettre dans une instruction `if` sont des *conditions booléennes*.

Un exemple de condition booléenne est un test de comparaison entre deux expressions par les opérateurs de comparaison ==, <, >, <=, >=, ! =. Ces opérateurs ont la signification suivante :

- x==y est vérifiée lorsque x est égal à y ;
- x<y est vérifiée lorsque x est strictement inférieur à y ;
- x>y est vérifiée lorsque x est strictement supérieur à y ;
- x<=y est vérifiée lorsque x est inférieur ou égal à y ;
- x>=y est vérifiée lorsque x est supérieur ou égal à y ;
- x !=y est vérifiée lorsque x est différent de y ;

Un nombre (entier, réel, etc.) pourra être considéré comme vrai ou faux avec la convention suivante :

- Une expression non nulle (différente de 0) est considéré comme vraie ;
- Une expression nulle (égale à 0) est considérée comme fausse.

Souvent, on donne la valeur 1 pour indiquer "vrai" et la valeur 0 pour indiquer "faux".

Exemple

```
int reponse;

puts("Tapez 1 pour conserver le message, 0 pour l'effacer");

scanf("%d", &reponse);

if (reponse) /* Si réponse est différent de zéro ! */
   conserver();
else
   effacer();
```

Dans cet exemple, les fonctions `conserver` et `effacer` doivent bien sûr avoir été préalablement conçues (voir le chapitre 6).

5.4.2 Opérations booléennes

a) Conjonction

La conjonction de deux expressions booléennes e_1 et e_2 est vérifiée lorsque e_1 et e_2 sont toutes deux vérifiées.

```
si (e1 et e2)
alors série d'instructions
fin si
```

En langage *C*, la conjonction est donnée par l'opérateur &&.

Exemple à la douane

```
char papiers, rad;

puts("Avez-vous vos papiers ? (y/n)");
papiers = getchar();
getchar();  /* getchar pour manger le retour chariot */
puts("Avez-vous quelque chose à déclarer ? (y/n)");
rad = getchar();

if (papiers == 'y' && rad == 'n')
   puts("O.K., vous pouvez passer. Bon voyage.");
```



Compléments

√ **Utilisation d'un `getchar` pour manger un retour chariot.** Lorsque l'utilisateur saisit un caractère ou une donnée par `getchar` ou `scanf`, il doit appuyer sur la touche *Entrée* (ou *retour chariot*) du clavier pour que le programme se poursuive. Le caractère retour chariot se trouve alors dans le flot de caractères en entrée du programme. Si l'on n'y prend garde, ce retour chariot sera lu par le `scanf` ou `getchar` suivant, provoquant un bug. On peut éliminer le retour chariot en faisant un `getchar` dans le vide comme dans l'exemple précédent.

b) Disjonction

La disjonction de deux expressions booléennes e_1 et e_2 est vérifiée lorsque l'une au moins de e_1 et e_2 est vérifiée.

```
si (e1 ou e2)
alors série d'instructions
fin si
```

En langage *C*, la disjonction est donnée par l'opérateur ||.

Exemple à la douane

```
char papiers, rad;

puts("Avez-vous vos papiers ? (y/n)");
papiers = getchar();
getchar();
puts("Avez-vous quelque chose à déclarer ? (y/n)");
rad = getchar();

if (papiers != 'y' || rad != 'n')
   puts("Attendez là s'il vous plait.");
```

c) Négation

La négation d'une condition *e* est vraie lorsque la condition *e* est fausse.

```
si (non e)
alors série d'instructions
fin si
```

En langage *C*, la négation est donnée par un **!** (point d'exclamation).

Exemple à la douane

```
char papiers, rad;

puts("Avez-vous vos papiers ? (y/n)");
papiers = getchar();
getchar();
puts("Avez-vous quelque chose à déclarer ? (y/n)");
rad = getchar();

if (!(papiers == 'y' && rad == 'n'))
   puts("Attendez là s'il vous plaît.");
```



Compléments

√ La **négation d'une conjonction**, comme !(A&&B) est égale à la **disjonction** des négations (!A)||(!B). De même, la négation d'une disjonction !(A||B) est égale à la conjonction des négations (!A)&&(!B). Cette loi s'appelle la *loi de Morgan*. (À méditer à tête reposée en prenant des exemples)

√ On peut composer les opérateurs !, &&, || à l'envie en créant des expressions comme (!A&&B||C&&D). Dans l'évaluation de l'expression, les priorités d'évaluation sont d'abord le !, ensuite le &&, puis le ||. Ainsi, l'expression (!A&&B||C&&D) est équivalente à (((!A)&&B)||(C&&D)). En cas de doute, il vaut toujours mieux mettre les parenthèses pour ne pas se tromper.

5.5 LE switch

Le switch permet de distinguer plusieurs cas selon les valeurs d'une variable. Le `if` permet de distinguer deux cas, alors que le `switch` permet de distinguer un grand nombre de cas.

Exemple de menu d'une base de données commerciale

```
char choix;

puts("Menu : faites un choix:\n");
puts("Afficher la liste des clients -----> a");
puts("Afficher les données d'un client --> b");
puts("Saisir un client ------------------> c");
puts("Quitter ---------------------------> d");

choix = getchar();

switch(choix)
{
   case 'a' : puts("Affichage de la liste des clients");
              /* mettre ici le code d'affichage des clients */
              break;
   case 'b' : puts("Affichage des données d'un client");
              puts("Veuillez entrer le nom du client");
              /* mettre ici le code de saisie et d'affichage */
              break;
   case 'c' : puts("Saisie des données du client");
              puts("Veuillez entrer les données");
              /* mettre ici le code de saisie des données */
              break;
   case 'd' :
              break;
   default : puts("Erreur de saisie du choix !");
}
```

Ici, la variable choix est un `char`. Un code différent est exécuté suivant les valeurs de la variable `choix`. Le cas `default`, qui est facultatif, correspond au code à exécuter si l'on n'entre dans aucun des autres cas. On peut aussi faire un `switch` avec des variables de type `int` au lieu de `char`. Dans ce cas, on ne trouve pas de quotes `' '` au niveau du `case`.



Ne pas oublier le `break` après le traitement de chaque cas. Cette erreur n'est pas détectée par le compilateur, mais en cas d'oubli du `break`, les instructions des cas suivants seraient aussi exécutées.

Exercices

5.1 (∗) Écrivez un programme qui lit deux variables au clavier et les affiche dans l'ordre croissant, quitte à les modifier.

5.2 (∗) Une entreprise X vend deux types de produits. Les produits de type A qui donnent lieu à une TVA à $5,5\%$, et les produits de type B, qui donnent lieu à une TVA à $19,6\%$. Écrire un programme qui lit au clavier le prix hors taxe d'un produit, saisit au clavier le type du produit et affiche le taux de TVA et le prix TTC du produit.

5.3 (∗) Écrivez un programme qui lit trois variables au clavier et affiche le maximum des trois.

5.4 (∗) Soit une équation du second degré $ax^2 + bx + c = 0$. Écrire un programme qui lit a, b, c au clavier et affiche les éventuelles solutions.

5.5 (∗) Écrire un programme qui saisit au clavier deux caractères `'+'` ou `'-'` et calcule le signe `'+'` ou `'-'` du produit.

5.6 (∗) Écrire un programme qui lit deux nombres entiers a et b et donne le choix à l'utilisateur :

1. de savoir si la somme $a + b$ est paire ;
2. de savoir si le produit ab est pair ;
3. de connaître le signe de la somme $a + b$;
4. de connaître le signe du produit ab.

Corrigés

5.1

```
int main(void)
{
  int n1, n2, temp;
  printf("Entrez deux nombres entiers : ");
  scanf("%d %d", &n1, &n2);
```

```
  if (n1 > n2)
    {
      temp = n1;
      n1 = n2;
      n2 = temp;
    }
  printf("Plus petit : %d ; plus grand : %d\n", n1, n2);
  return 0;
}
```

5.2

```
int main(void)
{
  float prix, tva, ttc;
  char type;
  printf("Entrez un prix : ");
  scanf("%f", &prix);
  printf("Produit de type A ou B ? ");
  getchar();
  type = (char) getchar();
  if (type == 'A')
    {
      tva = 5.5;
    }
  else
    {
      tva = 19.6;
    }
  ttc = prix * (1.0 + tva / 100.0);
  printf("Prix TTC : %f (TVA à %.1f %%)\n", ttc, tva);
  return 0;
}
```

5.3

```
int main(void)
{
  float a, b, c, max;
  printf("Entrez trois nombres réels : ");
  scanf("%f %f %f", &a, &b, &c);
  if (a < b)
    max = b;
  else
    max = a;
```

```
  if (max < c)
    max = c;
  printf("Le maximum des trois nombres est %f\n", max);
  return 0;
}
```

5.4

```
int main(void)
{
  float a, b, c, delta, racine1, racine2;
  printf("Entrez a, b et c (ax^2+bx+c) : ");
  scanf("%f %f %f", &a, &b, &c);
  delta = b * b - 4 * a * c;
  if (delta == 0.0)
    {
      racine1 = -b / (2 * a);
      printf("Ce polynôme possède une unique racine x = %f\n", racine1);
    }
  if (delta > 0.0)
    {
      racine1 = (-b - sqrt(delta)) / (2 * a);
      racine2 = (-b + sqrt(delta)) / (2 * a);
      printf("Ce polynôme possède deux racines x1 = %f et x2 = %f\n",
             racine1, racine2);
    }
  if (delta < 0.0)
    printf("Ce polynôme ne possède pas de racine réelle\n");
  return 0;
}
```

5.5

```
int main(void)
{
  char c1, c2, c;
  printf("Entrez deux caractères parmi + et - : ");
  scanf("%c %c", &c1, &c2);
  if (((c1 != '-') && (c1 != '+')) && ((c2 != '-') && (c2 != '-')))
    printf("Entrée incorrecte...\n");
  else
    {
      if (c1 == c2)
        c = '+';
```

```
      else
        c = '-';
      printf("Le signe du produit est %c\n", c);
    }
  return 0;
}
```

5.6

```
int main(void)
{
  int a, b;
  char choix;
  printf("Entrez deux nombres entiers : ");
  scanf("%d %d", &a, &b);
  printf("Tapez\n");
  printf("1 pour savoir si la somme est paire\n");
  printf("2 pour savoir si le produit est pair\n");
  printf("3 pour connaître le signe de la somme\n");
  printf("4 pour connaître le signe du produit\n");
  getchar();
  choix = (char) getchar();
  switch (choix)
    {
    case '1':
      if ((a + b) % 2 == 0)
        printf("La somme est paire\n");
      else
        printf("La somme est impaire\n");
      break;
    case '2':
      if ((a * b) % 2 == 0)
        printf("Le produit est pair\n");
      else
        printf("Le produit est impair\n");
      break;
    case '3':
      if (a + b >= 0)
        printf("La somme est positive\n");
      else
        printf("La somme est strictement négative\n");
      break;
    case '4':
      if (a * b >= 0)
        printf("Le produit est positif\n");
```

```
      else
        printf("Le produit est strictement négatif\n");
      break;
    default:
      printf("Choix non conforme...\n");
    }
  return 0;
}
```

Structuration d'un programme C

6

6.1 Qu'est-ce qu'un sous-programme ?

Lorsqu'un programme comprend de nombreuses lignes de code, il est difficile, voire impossible de mettre toutes les instructions les unes à la suite des autres dans le programme principal `main`. Le programme serait illisible, comprendrait trop de variables, etc. Pour cette raison, on décompose les problèmes en sous-problèmes et le programme en sous-programmes qui résolvent les sous-problèmes. En langage *C*, l'outil pour créer des sous-programmes est la notion de *fonction*.

6.2 Exemple de fonction C

Supposons que l'on ait plusieurs fois à calculer une expression, par exemple n^7 pour différentes valeurs d'un entier n. Il est malcommode de répéter la multiplication de n sept fois par lui-même et l'utilisation de la fonction `pow` de la bibliothèque `math.h` n'est pas recommandée car elle ne travaille pas sur le type `int`, mais sur le type `double`. Dans ce cas, on peut se fabriquer sa propre fonction :

```
#include <stdio.h>

int Puissance7(int a)     /* prototype de la fonction */
{
   int resultat;             /* variable locale */
   resultat = a*a*a*a*a*a*a;  /* calcul de a^7 */
   return resultat;              /* on renvoie le résultat */
}

/* Exemple d'utilisation dans le main : */

int main(void)
{
   int n, puiss7;           /* variables du main */
   printf("Veuillez entrer n");
   scanf("%d", &n);            /* lecture de n */
   puiss7 = Puissance7(n);   /* appel de la fonction */
   /* on récupère le résultat dans la variable puiss7 */
   printf("n puissance 7 est égal à %d\n", puiss7);
   return 0;
}
```

La fonction `Puissance7` a un *paramètre* : l'entier `a`. La fonction calcule a^7 et retourne le résultat pour qu'on puisse récupérer ce résultat dans la fonction `main`. Dans le `main`, on appelle la fonction `Puissance7` en passant en paramètre l'entier `n`. Sur cette ligne de code, **la valeur de `n` est recopié dans le paramètre `a` de la fonction** `Puissance7`. Le code de la fonction `Puissance7` est ensuite exécuté, calculant a^7. Dans le `main`, **la valeur retournée par la fonction `Puissance7` est récupérée par une affectation dans la variable** `puiss7`. En fin de compte, la variable `puiss7` contient la valeur n^7. On peut ensuite utiliser cette valeur, en l'occurence en l'affichant.

Remarque

La fonction `Puissance7` précédente effectue 6 multiplications. On peut donner une version qui ne fait que 4 multiplications (avec un gain d'efficacité) :

```
int Puissance7bis(int n) /* prototype de la fonction */
{
   int resultat, n2;       /* variables locales */
   n2 = n*n;                /* calcul de n2 */
   resultat = n2*n2*n2*n;  /* calcul de n7 */
   return resultat;           /* on renvoie le résultat */
}
```

6.3 Exemple de structuration d'un programme

Prenons un exemple très simple (peut-être trop simple) de programme, que nous décomposerons en sous-programmes. Il s'agit du programme qui lit un nombre x au clavier et calcule $f(x) = (x^3 - 2x + 1)\sin(3x + 1)$.

```
#include <stdio.h>
#include <math.h>   /* pour la fonction sinus */

int main()
{
   float x, y;  /* on calcule y=f(x) */
   puts("Veuillez taper une valeur de x :");
   scanf("%f", &x);
   y = (x*x*x-2*x+1)*sin(3*x+1);
   printf("on a : f(%f) = %f", x, y);
   return 0;
}
```

La décomposition du programme en fonctions donne :

```
#include <stdio.h>
#include <math.h>

/* Fonction Lire, permet de lire une valeur */
float Lire(void)  /* La fonction lire retourne un float */
{
   float d;  /* déclaration d'une variable locale d */
   puts("Veuillez taper une valeur :");
   scanf("%f", &d); /* on lit la valeur de d */
   return d;     /* on renvoie la valeur de d */
}

/* Fonction CalculF, calcule une valeur de la fonction */
float CalculF(float x)  /* renvoie f(x) pour x float */
{
   return (x*x*x-2*x+1)*sin(3*x+1);
}

/* Fonction Affiche, permet d'afficher une valeur */
void Affiche(float y)  /* affiche une valeur float y */
{
   printf("La valeur calculée est %f\n", y);
  /* une fonction void ne retourne rien */
}

/* Fonction main, programme principal */
int main()
{
   float x, y; /* on calcule y=f(x) */
   x = Lire();   /* appel de la fonction Lire */
   y = CalculF(x); /* calcul de f(x) */
   Affiche(y);  /* appel de la fonction Affiche */
   return 0;
}
```

Ici, la fonction `Lire` ne prend aucun paramètre (paramètre `void`). En fait, le résultat retourné par la fonction `Lire` ne dépend que de ce que l'utilisateur saisira au clavier et d'aucun autre paramètre. La fonction `CalculF` prend en paramètre un réel `x` et retourne $f(x)$. Notons qu'après le `return`, on ne trouve pas nécessairement une variable, mais on peut retourner toute une expression.

Dans cet exemple, la variable passée en paramètre dans le `main` porte le même nom, `x`, que le paramètre de la fonction `CalculF`. Cela ne pose pas de problème

mais il faut être conscient du fait que **le** `x` **dans le** `main` **et le** `x` **dans la fonction** `CalculF` **sont des variables différentes**, qui sont stockées dans des emplacements mémoire différents, même si **la valeur de la variable** `x` **du** `main` **est recopiée dans le paramètre** `x` **de** `CalculF`.

La fonction `Affiche` ne retourne aucune valeur. Elle effectue simplement des affichages et ne renvoie aucun résultat au `main`. Son type de retour est `void`. Une telle fonction est aussi appelée *procédure*.

L'intérêt de cette manière de structurer un programme est, même sur des tout petits programmes comme cet exemple, de décomposer le programme en phases (lecture, calcul, affichage...). Cela augmente la *modularité* car les fonctions `Lire` ou `Affiche` peuvent être réutilisées tel quel par d'autres programmes. On peut aussi facilement adapter la fonction `CalculF` pour calculer d'autres fonctions. Cela augmente aussi la lisibilité du code source.

Remarque. La fonction `main` pourrait aussi être réécrite :

```
int main()
{
   Affiche(CalculF(Lire()));
   return 0;
}
```

Ce style n'est cependant pas recommandé tant que les passages de paramètres et récupération de valeurs retournées ne sont pas parfaitement maîtrisés.

6.4 Forme générale d'une fonction *C*

6.4.1 Prototype

Le *prototype* d'une fonction, qui dit comment on doit l'utiliser, est de la forme :

```
typeRetour NomDeFonction(type1 nom1, type2 nom2,...)
```

avec :

- `typeRetour` qui est le type de la valeur retournée (ou renvoyée) par la fonction. Si `typeRetour` est `void`, la fonction ne retourne rien et on parle de *procédure*. Une procédure peut par exemple réaliser des affichages, écrire dans un fichier, etc.
- `NomDeFonction` qui est le nom de la fonction.
- `type1` et `nom1` qui sont respectivement le type et le nom du premier paramètre, `type2` et `nom2` le type et le nom du deuxième paramètre, etc. Notons que **la fonction peut comporter plusieurs paramètres**, qui sont séparés par des virgules, et

que l'on doit indiquer le type de chaque paramètre, même si plusieurs paramètres ont le même type (on doit alors répéter le type pour chaque paramètre).

Si la liste des paramètres est vide ou `void`, la fonction ne prend pas de paramètre. La fonction peut par exemple, prendre ses données au clavier, dans un fichier, etc.

6.4.2 Déclaration et définition

Une *déclaration de fonction* est le prototype de cette fonction suivi d'un point virgule- Une déclaration peut être répétée plusieurs fois dans le programme et indique de quelle sorte de fonction il s'agit (type de retour, types des paramètres...).

Une *définition de fonction* est le prototype suivi du corps de la fonction (suite de variables et d'instructions) entre accolades. La définition doit apparaître une seule fois dans le programme et indique ce que fait la fonction.

6.4.3 Variables locales

À l'intérieur de la définition d'une fonction, on trouve des déclarations de variables. Ces variables sont *locales*, c'est-à-dire qu'on ne peut les utiliser qu'à l'intérieur de cette fonction. Si l'on trouve une variable de même nom ailleurs dans le programme, il ne s'agit pas de la même variable, mais d'une variable homonyme.

6.5 Passage de paramètres par valeur

Lorsqu'on passe un paramètre à une fonction, la fonction ne peut pas modifier la variable. La variable est automatiquement recopiée, et la fonction travaille sur une copie de la variable. On appelle cette technique le *passage de paramètre par valeur*.

```
#include <stdio.h>

void NeModifiePas(int x)
{
   x = 2;  /* le x local est modifiée, pas le x du main */
}

int main(void)
{
   int x=1;
   NeModifiePas(x);
   printf("%d", x);  /* affiche 1 (valeur inchangée) */
}
```

Pour le moment, la seule manière que nous avons vu pour qu'une fonction transmette une valeur au `main` est pour cette fonction de retourner un résultat par `return`.

Avec cette technique, la fonction ne peut retourner qu'une seule valeur. Nous verrons plus loin la technique du passage de paramètres par adresse, qui permet à une fonction de modifier le contenu d'une variable du `main`, et donc de transmettre plusieurs valeurs au `main`.

Exercices

6.1 (∗) Écrire une fonction *C* qui affiche le code *ASCII* d'un caractère passé en paramètre. Écrire un programme principal qui saisit un caractère au clavier et affiche son code *ASCII*.

6.2 (∗) Écrire une fonction *C* qui calcule la moyenne de trois nombres passés en paramètre. Écrire le programme principal qui saisit trois nombres au clavier et affiche leur moyenne.

6.3 (∗) Écrire une fonction *C* qui **affiche** l'aire d'un triangle dont la base et la hauteur sont passées en paramètre. Écrire le programme principal qui saisit la base et la hauteur d'un triangle et affiche l'aire du triangle.

6.4 (∗) Soit un barême de l'impôt défini comme suit : pour un ménage X avec un revenu total R et un nombre n de membres du foyer, l'impôt est donné par :

- 10% de R si $\frac{R}{n} < 500$ euros ;
- 20% de R si $\frac{R}{n} \geq 500$ euros.

a) Écrire une fonction `Impot` qui calcule le montant de l'impôt en fonction de R et n.

b) Écrire une fonction `RevenuNet` qui calcule le revenu net d'un ménage après paiement de l'impôt en fonction de R et n. On pourra faire un programme principal qui saisit R et n au clavier, puis affiche l'impôt et le revenu net.

6.5 (∗∗) Soit la fonction mathématique f définie par $f(x) = \frac{(2x^3+3)(x^2-1)}{\sqrt{3x^2+1}}$.

a) Écrire une fonction *C* qui retourne la valeur de $f(x)$ pour un point x passé en paramètre.

b) Une approximation de la dérivée f' de la fonction f est donnée en chaque point x, pour h assez petit (proche de 0), par :

$$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h}.$$

Écrire une fonction *C* qui calcule une approximation de la dérivée f' de f en un point x entré au clavier. On passera la valeur de h en paramètre de la fonction.

c) La dérivée seconde de f est la dérivée de la dérivée. Écrire une fonction *C* qui calcule une approximation de la dérivée seconde f'' de f en un point x entré au clavier. On passera la valeur de h en paramètre de la fonction.

d) Écrire une fonction *C* qui détermine le signe de la dérivée seconde de f en fonction de x. On pourra faire un programme principal qui lit x au clavier et affiche le résultat.

e) Écrire une fonction *C* qui donne le choix à l'utilisateur d'afficher la valeur de la fonction f, de sa dérivée première ou de sa dérivée seconde en un point x lu au clavier.

Corrigés

6.1

```
void afficheASCII(char caractere)
{
  printf("Code ASCII = %d\n", caractere);
}
int main(void)
{
  char c;
  printf("Entrez un caractère : ");
  c = (char) getchar();
  afficheASCII(c);
  return 0;
}
```

6.2

```
float moyenne(float a, float b, float c)
{
  return (a + b + c) / 3.0;
}
int main(void)
{
  float n1, n2, n3;
  printf("Entrez trois nombres : ");
  scanf("%f %f %f", &n1, &n2, &n3);
  printf("La moyenne des nombres vaut %f\n", moyenne(n1, n2, n3));
  return 0;
}
```

6.3

```
void aire(float base, float hauteur)
{
  printf("Aire = %f\n", (base * hauteur) / 2.0);
}
int main(void)
{
  float b, h;
  printf("Entrez la base et la hauteur du triangle : ");
  scanf("%f %f", &b, &h);
  aire(b, h);
  return 0;
}
```

6.4

a)

```
float Impot(float R, int n)
{
  if (R / n < 500.0)
    return 0.1 * R;
  else
    return 0.2 * R;
}
```

b)

```
float RevenuNet(float R, int n)
{
  return R - Impot(R, n);
}
```

6.5

a)

```
float f(float x)
{
  return ((2 * x * x * x + 3) * (x * x - 1)) / sqrt(3 * x * x + 1);
}
```

b)

```
float fPrime(float x, float h)
{
  return (f(x + h) - f(x - h)) / (2 * h);
}
```

c)

```
float fSeconde(float x, float h)
{
  return (fPrime(x + h, h) - fPrime(x - h, h)) / (2 * h);
}
```

d)

```
int signe(float x)
{
  float h, dSeconde;
  printf("Donnez le pas d'approximation h : ");
  scanf("%f", &h);
  dSeconde = fSeconde(x, h);
  if (dSeconde < 0.0)
    return -1;
  if (dSeconde == 0.0)
    return 0;
  return 1;
}
```

e)

```
void choix()
{
  int c;
  float x, h;
  printf("Voulez-vous...\n");
  printf("1 la valeur de f\n");
  printf("2 la valeur de f'\n");
  printf("3 la valeur de f''\n");
  c = getchar();
  printf("x = ? ");
  scanf("%f", &x);
  switch (c)
    {
    case '1':
      printf("f(%f) = %f\n", x, f(x));
      break;
    case '2':
      printf("h = ? ");
      scanf("%f", &h);
      printf("f'(%f) = %f\n", x, fPrime(x, h));
      break;
    case '3':
      printf("h = ? ");
      scanf("%f", &h);
```

```
      printf("f''(%f) = %f\n", x, fSeconde(x, h));
      break;
    default:
      printf("Choix inconnu...\n");
    }
}
```

Structures 7

7.1 Déclaration d'une structure

Une *structure* est un type qui permet de stocker plusieurs données, de même type ou de types différents, dans une même variable de type structure. Une structure est composée de plusieurs champs, chaque champ correspondant à une donnée.

> ***Exemple***
>
> Voici la déclaration d'une structure `Point` qui contient trois champs x, y et z de type `float`.

```
struct point
{   /* déclaration de la structure */
    float x,y;  /* trois champs x, y, z */
    float z;    /* les champs se déclarent comme des variables */
                 /* mais on ne peut pas initialiser les valeurs */
};
```

La déclaration d'une variable de type `struct point` se fait ensuite comme pour une autre variable :

```
struct point P;  /* Déclaration d'une variable P */
```

7.2 Utilisation d'une structure

Une fois la variable déclarée, on accède aux données `x,y,z` du point `P` par un point. Ces données sont désignées dans le programme par `P.x,P.y,P.z`. Ces données `P.x,P.y,P.z`, ici de type `float`, sont traitées comme n'importe quelle autre donnée de type `float` dans le programme.

Notons que l'on pourrait rajouter d'autres données des types que l'on souhaite à la suite des données `x,y,z` dans la structure. Voici un exemple de programme avec une structure point.

```
#include <stdio.h>

struct point
{  /* déclaration de la structure */
```

```
   float x,y,z;
}; /* ne pas oublier le point-virgule */

int main(void)
{
   struct point P;
   puts("Veuillez entrer les coordonnées d'un point 3D :");
   scanf("%f %f %f", &P.x, &P.y, &P.z);
   puts("L'homothétie de centre O et de rapport 3");
   printf("appliquée à ce point donne :'');
   printf("(%.2f, %.2f, %.2f)\n", 3*P.x, 3*P.y, 3*P.z);
   return 0;
}
```

Pour éviter la répétition du mot `struct`, lors de la déclaration des variables de type `struc point`, on peut définir un raccourci par un `typedef` lors de la définition de la structure, pour donner un nouveau nom à ce type :

```
/* Définition d'un type Point3D */

typedef struct point
{                       /* déclaration d'un */
   float x,y,z;        /* nouveau type par typedef */
}Point3D;
```

La déclaration d'une variable de type `Point3D` se fait alors comme pour une variable d'un autre type en mettant d'abord le type puis l'identificateur de la variable :

```
Point3D P;     /* Déclaration d'une variable P de type Point3D */
```

Les données x, y et z du `Point3D P` sont toujours désignées par `P.x, P.y` et `P.z`.

Voyons maintenant un programme qui calcule l'addition (en tant que somme de vecteurs coordonnée par coordonnée) de deux `Point3D` entrés au clavier :

```
#include <stdio.h>

typedef struct point
{  /* déclaration de la structure */
   float x,y,z;
}Point3D;

/* la fonction suivante prend un Point3D en paramètre */
void Affiche(Point3D P)
{
```

```
   /* Affichage des champs : */
   printf("(%f, %f, %f)", P.x, P.y, P.z);
}

/* la fonction suivante retourne la structure */
Point3D SaisiePoint3D(void)
{
   Point3D P;  /* variable de type Point3D */
   printf("Entrez trois coordonnées séparées par des espaces\n");
   scanf("%f %f %f", &P.x, &P.y, &P.z); /* saisie des champs */
   return P;   /* on retourne la variable */
}

Point3D Addition(Point3D P1, Point3D P2)
{
   Point3D resultat;
   resultat.x = P1.x + P2.x; /* calcul des coordonnées */
   resultat.y = P1.y + P2.y; /* accès aux champs par un . */
   resultat.z = P1.y + P2.z;
   return resultat; /* la fonction retourne la structure */
}

int main(void)
{
   Point3D p1, p2, add;
   printf("Entrez les coordonnées de deux points");
   p1 = SaisiePoint3D();  /* on récupère la structure saisie */
   p2 = SaisiePoint3D();
   add = Addition(p1,p2);  /* appel de la fonction addition */
   printf("L'addition vaut : ");
   Affiche(add);
   printf("\n");
   return 0;
}
```

Exercices

7.1 (∗) Définir une structure `NombreRationnel` permettant de coder un nombre rationnel, avec numérateur et dénominateur. On écrira des fonctions de saisie, d'affichage, de multiplication et d'addition de deux rationnels. Pour l'addition, pour simplifier, on ne cherchera pas nécessairement le plus petit dénominateur commun.

7.2 (∗) Une menuiserie industrielle gère un stock de panneaux de bois. Chaque panneau possède une largeur, une longueur et une épaisseur en millimètres, ainsi que le type de bois qui peut être pin (code 0), chêne (code 1) ou hêtre (code 2).

a) Définir une structure `Panneau` contenant toutes les informations relatives à un panneau de bois.

b) Écrire des fonctions de saisie et d'affichage d'un panneau de bois.

c) Écrire une fonction qui calcule le volume en mètres cube d'un panneau.

7.3 (∗) Un grossiste en composants électroniques vend quatre types de produits :

- Des cartes mères (code 1) ;
- Des processeurs (code 2) ;
- Des barettes mémoire (code 3) ;
- Des cartes graphiques (code 4).

Chaque produit possède une référence (qui est un nombre entier), un prix en euros et des quantités disponibles.

a) Définir une structure `Produit` qui code un produit.

b) Écrire une fonction de saisie et d'affichage des données d'un produit.

c) Écrire une fonction qui permet à un utilisateur de saisir une commande d'un produit. L'utilisateur saisit les quantités commandées et les données du produit. L'ordinateur affiche toutes les données de la commande, y compris le prix.

Corrigés

7.1

```
typedef struct rationnel
{
  int numerateur, denominateur;
} NombreRationnel;
NombreRationnel Saisie()
{
  NombreRationnel n;
  printf("Entrez le numérateur et le dénominateur : ");
  scanf("%f %f", &n.numerateur, &n.denominateur);
  return n;
}

void Affichage(NombreRationnel n)
{
  printf("%f/%f\n", n.numerateur, n.denominateur);
}

NombreRationnel Multiplier(NombreRationnel p, NombreRationnel q)
{
  NombreRationnel r;
  r.numerateur = p.numerateur * q.numerateur;
  r.denominateur = p.denominateur * q.denominateur;
  return r;
}

NombreRationnel Additionner(NombreRationnel p, NombreRationnel q)
{
  NombreRationnel r;
  r.numerateur =
    p.numerateur * q.denominateur + q.numerateur * p.denominateur;
  r.denominateur = p.denominateur * q.denominateur;
  return r;
}
```

7.2

a)

```
typedef struct bois
{
  float largeur, longueur, epaisseur;
  char essence;
} Panneaux;
```

b)

```
Panneaux Saisie()
{
  Panneaux p;
  printf("Entrez la largeur, la longueur et l'épaisseur : ");
  scanf("%f %f %f", &p.largeur, &p.longueur, &p.epaisseur);
  printf("Entrez l'essence de bois : ");
  scanf("%c", &p.essence);
  return p;
}

void Affichage(Panneaux p)
{
  printf("Panneau en ");
  switch (p.essence)
    {
    case '0':
      printf("pin\n");
      break;
    case '1':
      printf("chêne\n");
      break;
    case '2':
      printf("hêtre\n");
      break;
    default:
      printf("inconnue\n");
    }
  printf("largeur = %f ; longueur = %f ; epaisseur = %f\n",
         p.largeur, p.longueur, p.epaisseur);
}
```

c)

```
float Volume(Panneaux p)
{
  return (p.largeur * p.longueur * p.epaisseur) / 1e9;
}
```

7.3

a)

```
typedef struct produit
{
  char type;
  unsigned int reference;
  float prix;
  unsigned int quantite;
} Produit;
```

b)

```
Produit Saisie()
{
  Produit p;
  printf("Entrez le code du produit : \n");
  scanf("%c", &p.type);
  printf("Entrez la référence : \n");
  scanf("%u", &p.reference);
  printf("Entrez le prix : \n");
  scanf("%f", &p.prix);
  printf("Entrez la quantité : \n");
  scanf("%u", &p.quantite);
  return p;
}

void Affichage(Produit p)
{
  printf("Type : %c\n", p.type);
  printf("Référence : %u\n", p.reference);
  printf("Prix : %f\n", p.prix);
  printf("Quantité : %u\n", p.quantite);
}
```

c)

```
void Commande()
{
  unsigned int qte;
  Produit p = Saisie();
  printf("Entrez la quantité commandée : ");
  scanf("%u", &qte);
  printf("\nRécapitulatif de la commande\n");
  Affichage(p);
  printf("Valeur de la commande : %.2f\n", p.prix * qte);
}
```

ITÉRATION 8

Une itération permet de répéter plusieurs fois une même série d'instructions, permettant de faire des récurrences ou de traiter de gros volumes de données.

8.1 BOUCLE while

Dans la boucle while, le programme répète un bloc d'instructions tant qu'une certaine condition est vraie. En langage algorithmique celà s'exprime par :

```
série d'instructions 1;     /* début du programme */
tant que (condition)
   faire série d'instructions 2;   /* instructions répétées */
fin tant que
série d'instructions 3;     /* suite du programme */
```

La condition s'appelle une *condition d'arrêt*. Lorsque la condition d'arrêt devient fausse, le programme sort de la boucle tant que et passe à la suite du programme. Si le programme est bien conçu, la série d'instructions 2 doit rendre la condition d'arrêt fausse à un moment donné, sans quoi, le programme se poursuit indéfiniment dans une boucle sans fin (on dit que le programme *boucle*).

Prenons un exemple. Supposons que l'on veuille faire une fonction qui calcule n^k, où n est un entier et k est un exposant entier saisi au clavier. Le résultat doit être un entier. Il ne vaut mieux pas utiliser la fonction pow de math.h car celle-ci passe par du calcul en nombres réels flottants, ce qui pose des problèmes d'arrondi.

```
int Puissance(int n, int k)
{
   int i = 0;  /* initialisation  obligatoire */
   int resultat = 1; /* initialisation à 1 (calcul du produit) */
   while (i < k)   /* boucle tant que */
      {
         resultat = resultat*n;
         i = i+1;   /* progression  obligatoire */
      }
   return resultat;
}
```

Au départ, la variable i vaut 0 (initialisation). Si k est strictement positif, au départ, la condition d'arrêt i<k est vraie, et on rentre dans le while. À chaque exécution des instructions du while, l'instruction i=i+1 est exécutée, ce qui augmente la valeur

de la variable `i` (on parle d'*incrémentation* lorsqu'on augmente de 1 la valeur d'une variable). La condition d'arrêt est alors retestée avant d'effectuer l'itération suivante. Comme la variable `i` augmente à chaque itération, au bout d'un moment la condition d'arrêt `i<k` devient fausse, et la boucle se termine ; on passe à la suite du programme, en l'occurrence l'instruction `return`. Si jamais `k` est égal à 0, la condition d'arrêt est fausse dès le départ, le programme ne rentre pas du tout dans le `while`, et le résultat vaut 1, ce qui est bien égal à n^k avec $k = 0$.

De cette manière, on accumule les multiplications par n dans la variable `resultat`, qui au départ vaut 1. Au fil des itérations, la variable `i` augmente et finit par devenir égale à `k` au bout d'exactement `k` itérations. La condition d'arrêt `i < k` devient alors fausse, et on sort de la boucle. La variable `resultat` vaut bien n^k, produit de `n` par lui-même `k` fois.



Attention à ne pas faire une multiplication en trop ou en moins : la variable *i* commence à 0 et on met une condition < et non pas <=. La multiplication par `n` se produit donc exactement `k` fois.



Il s'agit d'une condition **tant que** et non pas d'une condition **juqu'à ce que**. Les instructions sont répétées tant que la condition d'arrêt est vraie, et le programme sort de la boucle quand la condition devient fausse, et non pas le contraire.

Notons que les instructions dans le `while` doivent constituer un bloc, c'est-à-dire qu'elles doivent être encadrées par des accolades `{}`, sauf s'il n'y a qu'une seule instruction, auquel cas les accolades sont facultatives.

8.2 BOUCLE `for`

On voit qu'en général, la structure d'une itération `while` est de la forme :

```
initialisation;
tant que (condition)
    faire
      {
          série d'instructions;
          progression;
      }
```

Il y a en *C* une syntaxe plus commode pour faire cet ensemble de choses, c'est la boucle for :

```
for (initialisation ; condition ; progression)
   {
      série d'instructions;
   }
```

Ainsi, le programme ci-dessus qui calcule n^k peut être réécrit :

```
int PuissanceBis(int n, int k)
{
   int i;             /* plus d'initialisation ici */
   int resultat = 1; /* initialisation à 1 (calcul du produit) */
   for (i=0 ; i < k ; i=i+1)  /* boucle for */
      {
         resultat = resultat*n;
      }
   return resultat;
}
```

Au début de la boucle `for`, l'instruction d'initialisation `i=0` est exécutée. La condition d'arrêt est ensuite testée. Si la condition d'arrêt est vraie, les instructions dans le `for` sont exécutées, après quoi l'instruction de progression `i=i+1` est exécutée. La condition d'arrêt est re-testée ensuite avant l'itération suivante, etc. **tant que** la condition d'arrêt est vraie.

Notons que dans cet exemple les accolades dans le `for` sont facultatives, puisqu'il n'y a qu'une seule instruction dans la boucle `for`.



Compléments

√ L'instruction `i=i+1` s'appelle *incrémentation* de la variable `i`. Une incrémentation augmente de 1 la valeur de la variable. Il y a en *C* une manière plus compacte d'écrire une incrémentation, qui est d'écrire `i++` à la place de `i=i+1`.

Exercices

Boucles `while`

8.1 (∗) Écrire une fonction qui calcule la factorielle $n!$ d'un entier n passé en paramètre. (Rappel : $n! = 1 \times 2 \times \cdots \times n$)

8.2 (∗) Écrire une fonction qui calcule la somme :

$$s = 1 + 2^3 + 3^3 + \cdots + n^3$$

en fonction de n.

8.3 (∗∗) On veut écrire un programme de machine à voter. On fait voter des utilisateurs tant qu'il y en a entre un candidat A et un candidat B. À chaque fois, l'ordinateur demande s'il doit continuer. À la fin, l'ordinateur doit afficher les pourcentages de voix et le vainqueur. Le programme doit être fiable à 100% et ne doit pas permettre d'erreur de saisie.

8.4 (∗∗)

a) Écrire une fonction qui calcule le nombre de chiffres en base 10 d'un nombre n.

b) Écrire une fonction qui calcule le $i$ème chiffre en base 10 d'un nombre n. Les entiers i et n sont entrés au clavier. On supposera que les chiffres sont numérotés à l'envers (le chiffre des unités est le numéro 0, le chiffre des dizaines est le numéro 1...)

Boucles `for`

8.5 (∗) Écrire une fonction qui affiche la valeur en degrés Celsius correspondant aux valeurs 0, 10, 20,..., 300 degrés Fahrenheit. On utilisera la formule de conversion :

$$C = \frac{5}{9}(F - 32)$$

8.6 (∗) Écrire une fonction qui renvoie la plus grande puissance de 2 inférieure à la constante $C = 2\,426\,555\,645$.

8.7 (∗) Une suite $(u_n)_{n\in\mathbb{N}}$ est définie par récurrence :

$$u_0 = 1 \text{ et } u_{n+1} = 2 * \sqrt{u_n + 1} - 1 \text{ pour } n \geq 0$$

Donner un algorithme pour calculer u_n, le nombre $n \geq 0$ étant saisi au clavier.

8.8 (∗∗) Soit $f(i) = \dfrac{150\sin(i)}{(i+1)}$ pour $n \in \mathbb{N}$.

a) Écrire une fonction qui renvoie le maximum des $f(i)$ pour i entier entre 0 et $N - 1$, où N est un entier passé en paramètre.

b) Écrire une fonction qui compte le nombre de valeurs $f(i)$ comprises entre $-a$ et $+a$, avec i entre 0 et $N - 1$, pour a réel passé en paramètre.

8.9 (∗∗) La suite de Fibonacci est définie par récurrence comme suit :

$$u_0 = 1,\ u_1 = 1,\ \text{et } u_n = u_{n-1} + u_{n-2} \text{ pour } n \geq 2$$

Écrire une fonction qui calcule le $n$ième terme de la suite de Fibonacci.

Corrigés

8.1

```
int fact(int n)
{
  int f = 1;
  while (n > 0)
    {
      f = f * n;
      n = n - 1;
    }
  return f;
}
```

8.2

```
int cube(int n)
{
  int s = 0;
  while (n > 0)
    {
      s = s + n * n * n;
      n = n - 1;
    }
  return s;
}
```

8.3

```
void voter()
{
  int nVotantA = 0, nVotantB = 0;
  char c;
  int stop = 0, choix;                  /* 0 <-> faux; !0 <-> vrai */
  float voix;
  while (!stop)
    {
      choix = 0;
      printf("On continue (o/n) ? ");
      while (!choix)
        {
          c = (char) getchar();
          choix = (c == 'o') || (c == 'n');
        }
```

```
        if (c == 'o')
          {
            choix = 0;
            printf("Vote pour A ou pour B ? ");
            while (!choix)
              {
                c = getchar();
                choix = (c == 'A') || (c == 'B');
              }
            if (c == 'A')
              nVotantA = nVotantA + 1;
            else
              nVotantB = nVotantB + 1;
          }
        else
          stop = !stop;
      }
    if (nVotantA + nVotantB > 0)
      {
        voix = (100.0 * nVotantA) / (float) (nVotantA + nVotantB);
        printf("A a obtenu %f %% des voix et B a obtenu %f %% des voix\n",
               voix, 100 - voix);
        if (nVotantA > nVotantB)
          printf("A est élu\n");
        if (nVotantA < nVotantB)
          printf("B est élu\n");
        if (nVotantA == nVotantB)
          printf("Aucun élu\n");
      }
}
```

8.4

a)

```
int base10(int n)
{
  int nombre = 0;
  while (n > 0)
    {
      n = n / 10;
      nombre = nombre + 1;
    }
  return nombre;
}
```

b)

```
int base10i(int n, int i)
{
  while (i > 0)
    {
      n = n / 10;
      i = i - 1;
    }
  return n % 10;
}
```

8.5

```
void celsius()
{
  int i;
  for (i = 0; i <= 300; i = i + 10)
    {
      printf("%d degrés Fahrenheit donnent %.2f degrés Celsius\n", i,
             (5 * (i - 32.0) / 9.0));
    }
}
```

8.6

```
#define C 2426555645
unsigned int puissance2()
{
  unsigned int n;
  int i = 0;
  for (n = C; n / 2 > 0; n = n / 2)
    i = i + 1;
  return i;
}

int main()
{
  printf("%u\n", puissance2());
  return 0;
}
```

8.7

```
float un(unsigned int n)
{
  float u = 1.0;
  unsigned int i;
  for (i = 1; i <= n; i++)
    u = 2 * sqrt(u + 1) - 1;
  return u;
}
```

8.8

a)

```
double f(int n)
{
  return (150 * sin(n)) / (double) (1 + n);
}
double max(int N)
{
  int i;
  double fi, maxi = f(0);
  for (i = 1; i < N; i++)
    {
      fi = f(i);
      if (fi > maxi)
        maxi = fi;
    }
  return maxi;
}
```

b)

```
int nombre(int N, double a)
{
  int i, nb;
  double fi;
  nb = 0;
  for (i = 0; i < N; i++)
    {
      fi = f(i);
      if (fabs(fi) <= a)
        nb++;
    }
  return nb;
}
```

8.9

```
int fibonacci(int n)
{
  int u = 1, v = 1;
  int i;
  for (i = 2; i <= n; i++)
    {
      v = u + v;
      u = v - u;
    }
  return v;
}
```

Partie 2

Structures séquentielles

TABLEAUX 9

Un tableau permet de mémoriser plusieurs données du même type. Contrairement aux variables simples, les tableaux permettent de stocker des données nombreuses en mémoire centrale. La différence avec les structures est que les données du tableau ne doivent pas toutes être déclarées à la main dans le code ; la déclaration du tableau en une ligne suffit pour déclarer toutes les données. En revanche, toutes les données du tableau doivent être de même type (même si le type des éléments du tableau peut être un type complexe comme une structure).

9.1 DÉCLARATION D'UN TABLEAU

On déclare un tableau (statique) par

```
typeElements nomTableau[NOMBRE_ELEMENTS];
```

où `typeElements` est le type des éléments du tableau, `nomTableau` est le nom du tableau, et `NOMBRE_ELEMENTS` est **une constante** indiquant le nombre d'éléments du tableau.
Par exemple, pour déclarer un tableau de 100 entiers appelé `tab` :

```
int tab[100];
```

Pour déclarer un tableau de 150 caractères appelé `chaine` ;

```
char chaine[150];
```

Dans une telle déclaration, le nombre d'éléments du tableau est **obligatoirement une constante**. Le nombre d'éléments du tableau peut être défini dans une directive `#define` ce qui augmente la souplesse du programme (on peut alors changer le nombre d'éléments en changeant au niveau du `#define`). Nous verrons dans le chapitre sur l'allocation dynamique comment créer des tableaux dont le nombre d'emplacements mémoire est donné par une variable.

9.2 ACCÈS AUX ÉLÉMENTS

Les éléments d'un tableau sont comme des cases rangées successivement dans la mémoire centrale. Les éléments d'un tableau sont numérotés par des indices. Par exemple, les éléments du tableau `tab` déclaré ci-dessus, qui comporte 100 éléments, ont des indices 0, 1, 2, . . . , 99.

tab[0]	tab[1]	tab[2]	tab[3]	tab[4]	...
0	1	2	3	4	...



Les indices des éléments d'un tableau commencent à 0 et non pas à 1. Par conséquent, les éléments d'un tableau à N éléments ont leurs indices allant de 0 à $N-1$. L'accès à un élément d'indice supérieur ou égal à N provoquera systématiquement un résultat faux, et généralement une erreur mémoire (erreur de segmentation). On parle de dépassement de tableaux.

L'élément d'indice 3 dans le tableau `tab` est noté `tab[3]`. Plus généralement, l'élément d'indice `i` dans le tableau `tab` est noté `tab[i]`. L'indice `i` doit impérativement être un nombre entier positif ou nul, mais ça peut être le résultat de toute une expression comme dans `tab[3*m+2]`, où m est un entier.

9.3 Nombre d'éléments fixé

Le programme suivant permet de mémoriser différentes valeurs saisies au clavier et de les réafficher dans l'ordre où elles ont été saisies. Le nombre de valeurs, ou nombre d'éléments du tableau, est fixé à 15.

```
#include <stdio.h>

#define NB_ELEM 15  /* Nombre d'éléments du tableau */

/* ******** Fonction Affichage ********** */
/*          Affiche un tableau            */

void Affichage(float tableau[NB_ELEM])
{
   int i;
   for (i=0 ; i<NB_ELEM ; i++)
   {
      printf("l'élément numéro %d vaut %f\n", i, tableau[i]);
   }
}

/* ********* Fonction main ********** */
/* Lit un tableau et le fait afficher */

int main()
{
   int i;   /* indice */
   float tableau[NB_ELEM];   /* déclaration du tableau */
   for (i=0 ; i<NB_ELEM ; i++)
```

```
    {
       printf("Entrez l'élément %d : ", i);
       scanf("%f", &tableau[i]);   /* lecture d'un élément */
    }
    Affichage(tableau);   /* Affichage du tableau */
    return 0;
}
```

9.4 Nombre d'éléments variable borné

Le programme suivant permet aussi de lire des éléments au clavier et de les réafficher, mais cette fois le nombre d'éléments est lu au clavier. Ce nombre d'éléments doit toutefois rester inférieur à une valeur maximale constante fixée. L'idée est de n'utiliser qu'une partie du tableau. On réserve des emplacements mémoires pour `NB_ELEM_MAXI` éléments dans le tableau, mais il n'est pas indispensable d'utiliser toutes les cases du tableau. Par contre, le nombre de cases utilisées doit rester inférieur ou égal à `NB_ELEM_MAXI`.

```
#include <stdio.h>

#define NB_ELEM_MAXI 100  /* Nombre maximum d'éléments */
                           /* du tableau */

/* ******** Fonction Affichage ********** */
/* Affiche un tableau de taille n         */

void Affichage(float tableau[], int n)
{
    int i;
    for (i=0 ; i<n ; i++)
    {
       printf("l'élément numéro %d vaut %f\n", i, tableau[i]);
    }
}

/* ********* Fonction main ********** */
/* Lit un tableau et le fait afficher */

int main()
{
    int n, i;   /* nombre d'éléments et indice */
    float tableau[NB_ELEM_MAXI];   /* déclaration du tableau */
    printf("Entrez le nombre d'éléments à taper : ");
```

```
    /* lecture du nombre d'éléments au clavier (variable) */
    scanf("%d", &n);
    if (n > NB_ELEM_MAXI)
    {    /* test d'erreur */
       puts("Erreur, nombre trop grand !");
       return 1;
    }
    for (i=0 ; i<n ; i++)
    {    /* n éléments */
       printf("Entrez l'élément %d : ", i);
       scanf("%f", &tableau[i]);    /* lecture d'un élément */
    }
    Affichage(tableau, n);    /* Affichage du tableau */
    return 0;
}
```

Le nombre maximum NB_ELEM_MAXI, qui correspond au nombre d'emplacements mémoire disponibles, est parfois appelé *taille physique* du tableau. Le nombre n d'éléments du tableau effectivement utilisés est la *taille logique* du tableau.

On pourrait aussi lire les éléments du tableau dans une fonction. Il est impossible de retourner un tableau (sauf s'il a été alloué dynamiquement comme expliqué au chapitre 12). Par contre, on peut modifier les éléments d'un tableau passé en paramètre pour les initialiser.

```
#include <stdio.h>

#define NB_ELEM_MAXI 100  /* Nombre maximum d'éléments */

/* ******** Fonction de saisie ********** */
/* Saisie d'un tableau au clavier         */
/* Le nombre d'éléments est retourné      */

int SaisieTableau(float tableau[NB_ELEM_MAXI])
{
    int n, i;
    puts("Entrez le nombre d'éléments du tableau : ");
    scanf("%d", &n);
    if (n > NB_ELEM_MAXI)
       {
          puts("Erreur : le tableau est trop petit");
          return -1;  /* retour d'un code d'erreur */
       }
    puts("Entrez un à un les éléments");
    for (i=0 ; i<n ; i++)
```

```
      scanf("%f", &tableau[i]);
   return n;
}

int main()
{
   int n;    /* nombre d'éléments */
   float tableau[NB_ELEM_MAXI];    /* déclaration du tableau */
   n = SaisieTableau(tableau);
   if (n > 0)  /* Test d'erreur */
      Affichage(tableau, n);    /* Affichage du tableau */
   return 0;
}
```



En général, on ne peut pas modifier les variables passées en paramètre d'une fonction (passage par valeur). Cependant, pour un tableau, on peut modifier les éléments d'un tableau passé en paramètre. Ceci est lié au fait, que nous comprendrons plus précisément au chapitre 12, que le type tableau est en fait une adresse à laquelle se trouve de la mémoire dans laquelle sont stockés les éléments.

9.5 Initialisation lors de la déclaration

Comme les autres types de variables, les éléments du tableau peuvent être initialisés lors de la déclaration du tableau. On met pour celà les valeurs des éléments entre accolades { } séparés par des virgules.

```
#include <stdio.h>

int main(void)
{
   int tab[5] = {3, 56, 21, 34, 6}; /* avec point-virgule */

   for (i=0 ; i<5 ; i++)  /* affichage des éléments */
      printf("tab[%d] = %d\n", i, tab[i]);
   return 0;
}
```

Exercices

9.1 (∗) Écrire une fonction qui prend en paramètre un tableau d'entiers et son nombre d'éléments, et qui affiche les éléments d'indice impair.

9.2 (∗) Écrire une fonction qui prend en paramètre un tableau d'entiers et calcule le maximum de tous les éléments du tableau.

9.3 (∗) Écrire une fonction qui prend en paramètre un tableau d'entiers et calcule la somme des éléments.

9.4 (∗) Écrire une fonction qui prend en paramètre un tableau d'entiers, son nombre d'éléments n et un entier m, et qui retourne 1 s'il y a un élément égal à m dans le tableau, 0 sinon. (Le type de retour de la fonction doit être *char*).

9.5 (∗∗) Écrire une fonction qui prend en paramètre un tableau de taille `NB_MAX`, son nombre d'éléments `n < NB_MAX`, un entier `i` ≤ `n`, et un entier `m`. La fonction doit insérer l'élément `m` en position `i` dans le tableau (sans supprimer d'élément).

9.6 (∗ ∗ ∗)

a) Écrire une fonction qui initialise chaque élément `tab[i]` d'un tableau `tab` passé en paramètre à la valeur 2^i.

b) En utilisant le a), écrire une fonction qui lit au clavier des caractères égaux à 0 ou 1 et calcule le nombre que ces chiffres représentent en binaire. On pourra faire afficher en base 10 le résultat.

Corrigés

9.1

```
#define NB_ELEM_MAXI 100
void afficheImpair(int t[NB_ELEM_MAXI], int taille)
{
  int i;
  for (i = 1; i < taille; i += 2)
    printf("t[%d] = %d\n", i, t[i]);
}
```

9.2

```
#define NB_ELEM_MAXI 100
int max(int t[NB_ELEM_MAXI])
{
  int i;
  int max = t[0];
  for (i = 1; i < NB_ELEM_MAXI; i++)
    {
      if (max < t[i])
        max = t[i];
    }
  return max;
}
```

9.3

```
#define NB_ELEM_MAXI 100
int somme(int t[NB_ELEM_MAXI])
{
  int i;
  int somme = t[0];
  for (i = 1; i < NB_ELEM_MAXI; i++)
    somme += t[i];
  return somme;
}
```

9.4

```
#define NB_ELEM_MAXI 100
char recherche(int t[NB_ELEM_MAXI], int taille, int m)
{
  int i;
  for (i = 0; i < taille; i++)
    {
      if (t[i] == m)
        return 1;
    }
  return 0;
}
```

9.5

```
#define NB_MAX 100
void insertion(int t[NB_MAX], int n, int i, int m)
{
  int j = n;
  while (j > i)
```

```
    {
      t[j] = t[j - 1];
      j--;
    }
  t[i] = m;
}
```

9.6

```
#define NB_MAX 31
```

a)

```
void remplir(unsigned int t[NB_MAX])
{
  int i = 1;
  t[0] = 1;
  while (i < NB_MAX)
    {
      t[i] = t[i - 1] * 2;
      i++;
    }
}
```

b)

```
void clavier()
{
  char c;
  int choix, i = 0;
  unsigned int n;
  unsigned int t[NB_MAX];
  remplir(t);
  while (!0)
    {
      choix = 0;
      printf("Entrez 0 ou 1 : ");
      while (!choix)
        {
          c = getchar();
          choix = ((c == '0') || (c == '1'));
        }
      if (c == '1')
        n += t[i];
      i++;
      printf("Valeur courante : %d\n", n);
    }
}
```

FICHIERS TEXTE 10

10.1 QU'EST-CE QU'UN FICHIER TEXTE ?

Dans un ordinateur, il y a principalement deux sortes de mémoire : la mémoire centrale et la mémoire disque. Les données stockées en mémoire centrale sont les variables des différents programmes et ne durent que le temps de l'exécution d'un programme. Les tableaux stockent les données en mémoire centrale. Pour mémoriser des données de manière permanente, il faut les stocker sur un disque (ou un périphérique tel qu'une clef USB).

Un *fichier* est une série de données stockées sur un disque ou dans un périphérique de stockage. Un *fichier texte* est un fichier qui contient du texte *ASCII*.

On appelle *lecture* dans un fichier le transfert de données du fichier vers la mémoire centrale. La lecture dans un fichier texte est analogue à la lecture au clavier : le texte vient du fichier au lieu de venir du clavier.

On appelle *écriture* dans un fichier le transfert de données de la mémoire centrale vers le fichier. L'écriture dans un fichier texte est analogue à l'écriture à l'écran : le texte va dans le fichier au lieu de s'afficher.

Le but de ce chapitre est de voir comment un programme *C* peut lire et écrire des données dans un fichier texte.

10.2 OUVERTURE ET FERMETURE D'UN FICHIER TEXTE

Pour pouvoir utiliser les fichiers texte, on doit inclure la bibliothèque d'entrées-sorties :

```
#include<stdio.h>
```

Pour lire ou écrire dans un fichier texte, nous avons besoin d'un *pointeur de fichier*. Le pointeur de fichier nous permet de désigner le fichier dans lequel nous souhaitons lire ou écrire. Un pointeur de fichier est de type *FILE* `*`. On déclare un tel pointeur comme toute autre variable :

```
FILE *fp;  /* déclaration d'un pointeur de fichier fp */
```

Avant de pouvoir écrire ou lire dans le fichier, il faut lier le pointeur de fichier à un fichier sur le disque. On appelle cette opération l'*ouverture du fichier*. L'ouverture se fait avec la fonction `fopen`, qui prend en paramètre le nom du fichier sur le disque.

Prenons le cas de l'ouverture d'un fichier en lecture, c'est-à-dire qu'on ouvre le fichier uniquement pour pouvoir lire des données dedans.

```
FILE *fp;
fp = fopen("monfichier.txt","r");
/* (exemple de chemin relatif : répertoire local) */
```

```
FILE *fp;
fp = fopen("/home/remy/algo/monfichier.txt","r");
/* (exemple de chemin absolu sous Unix ou Linux) */
```

```
FILE *fp;
fp = fopen("C:\\remy\\algo\\monfichier.txt","r");
/* (exemple de chemin absolu sous Windows) */
```

Le premier paramètre est le nom du fichier, ou plus exactement le chemin vers le fichier dans l'arborescence des répertoires. Dans le cas d'un chemin relatif, le chemin part du répertoire de travail (donné par `pwd` sous *Unix/Linux*). Dans le cas d'un chemin absolu, le chemin part de la racine de l'arborescence des répertoires (la partition / sous *Unix* ou *Linux* ou une partition *C* :, *D* :, etc... sous *Windows*). Notons que le caractère \ (*antislash*) est réprésenté dans les chaînes de caractères par un caractère spécial (au même titre que `'\n'` pour le retour à la ligne). Ce caractère spécial est noté `'\\'`. Ceci précisément car le \ est utilisé dans la désignation des caractères spéciaux comme `'\n'`, `'\%'`, etc.
Le deuxième paramètre de la fonction `fopen` est le *mode*, et `"r"` signifie “fichier ouvert en lecture seule”. Les différents modes possibles pour un fichier texte sont :

- `"r"` : mode lecture seule. Le fichier est ouvert à son début prêt à lire les données. Toute tentative d'écriture dans le fichier provoque une erreur de segmentation.
- `"w"` : mode écriture seule. Le fichier est initialement vide. Si le fichier existait déjà avant l'appel de `fopen`, il est écrasé et les données qui se trouvaient dans le fichier sont perdues. Après l'appel de `fopen`, le fichier est prêt pour l'écriture de données. Toute tentative de lecture provoque une erreur de segmentation.
- `"a"` : mode ajout. Le fichier n'est pas écrasé mais est prêt à écrire à la suite des données existantes. Toute tentative de lecture provoque une erreur de segmentation.
- `"r+"` : mode lecture-écriture. Le fichier est prêt pour lire et écrire au début du fichier. Le fichier n'est pas écrasé.
- `"w+"` : mode lecture-écriture. Le fichier est écrasé.
- `"a+"` : mode lecture-écriture. Le fichier n'est pas écrasé mais est prêt pour écrire à la suite des données existantes.

Les trois derniers modes (lecture-écriture) ne sont pas souvent utilisés pour les fichiers texte, mais nous verrons dans un autre chapitre comment les utiliser dans les fichiers binaires.

La fonction `fopen` retourne le pointeur `NULL` en cas d'erreur d'ouverture de fichier (fichier inexistant, erreur dans le nom de fichier, ou permissions, en lecture, écriture, selon le cas, insuffisantes pour accéder au fichier au niveau du système). Le pointeur `NULL` est très souvent utilisé comme code d'erreur pour les fonctions retournant un pointeur.

Après avoir utilisé un fichier, il faut le refermer en utilisant `fclose`. La fonction `fclose` prend en paramètre le pointeur de fichier et ferme le fichier.



En cas d'oubli de l'appel à `fclose`, certaines données risquent de ne pas être transférées entre le fichier et la mémoire centrale (ceci en raison de l'existance d'une “mémoire tampon”, en anglais *buffer*, où les données sont stockées temporairement en attendant leur transfert définitif ; l'appel à `fclose` provoque le transfert du buffer).

10.3 Lire et écrire des données formatées

10.3.1 Lire des données formatées

Pour lire dans un fichier, le fichier doit préalablement avoir été ouvert en mode `"r"`, `"r+"`, `"w+"`, ou `"a+"`.

Pour lire des données numériques (des nombres) ou autres dans un fichier texte, on peut utiliser la fonction `fscanf`, qui est analogue à `scanf` sauf qu'elle lit dans un fichier au lieu de lire au clavier. La fonction `fscanf` prend pour premier paramètre le pointeur de fichier, puis les autres paramètres sont les mêmes que ceux de `scanf` (la chaîne de format avec des `%d,%f,...`, puis les adresses des variables avec des `&`). La fonction `fscanf` **retourne le nombre de variables effectivement lues**, qui peut être inférieur au nombre de variables dont la lecture est demandée en cas d'erreur ou de fin de fichier. Ceci permet de détecter la fin du fichier ou les erreurs de lecture dans in `if` ou dans un `while`.

Exemple

Chargement d'un fichier d'entiers en mémoire centrale.
On suppose que dans un fichier texte sont écrits des nombres entiers séparés par des espaces :

```
10 2 35 752 -5 4 -52 etc.
```

Nous allons *charger* ces entiers en mémoire centrale (c'est-à-dire les mettre dans un tableau), puis nous allons afficher le tableau.

```
#include <stdio.h>
#include <stdlib.h>  /* pour utiliser la fonction exit */

#define NB_ELEM_MAX 100

int ChargeFichier(int tableau[NB_ELEM_MAX])
{
   FILE *fp;
   int i=0;
   /* ouverture du fichier : */
   fp = fopen("monfichier.txt", "rt");
   if (fp ==NULL)  /* gestion d'erreur */
      {
         puts("Erreur d'ouverture de fichier :");
         puts("Fichier inexistant ou permissions insuffisantes");
         exit(1); /* termine le programme avec code d'erreur */
      }
   /* on peut mettre des appels de fonctions comme fscanf dans
      une condition. Ici, fscanf retourne 1 en cas de succès */
   while (i < NB_ELEM_MAX && fscanf(fp, "%d", &tableau[i])==1)
         i++;   /* incrémentation : pareil que i=i+1 */
   fclose(fp);   /* fermeture du fichier */
   return i;  /* on retourne le nombre d'éléments lus */
}

void Affiche(int tableau[], int n)
{
   int i;
   for (i=0 ; i<n ; i++)
      printf("tableau[%d] = %d\n", i, tableau[i]);
}

int main()
{
   int tab[NB_ELEM_MAX];
   int n;
   n = ChargeFichier(tab);
   Affiche(tab, n);
   return 0;
}
```

La fonction `exit` de la bibliothèque `stdlib.h` permet de terminer le programme à l'endroit où elle est appelée (avec éventuellement un code d'erreur passé en paramètre qui est transmis au système). Ceci ne doit pas être confondu avec le `return` qui sort de la fonction en cours mais ne termine pas le programme (sauf si la fonction en

cours est le `main`). Dans le `main`, un `return` est équivalent à un `exit` : il termine le programme. Dans une autre fonction que le `main`, le `return` fait passer l'exécution du programme à la fonction qui a appelé la fonction en cours (par exemple le `main`), juste après l'appel de la fonction en cours, mais l'exécution du programme se poursuit.

Notons la condition dans le `while` qui teste la valeur retournée par `fscanf`. Dans cet exemple, on demande à lire une seule valeur dans le `fscanf`. La fonction `fscanf` doit retourner 1 en cas de succès. À la fin du fichier, la lecture de la valeur échoue dans le `fscanf`, et la fonction `fscanf` retourne alors une valeur différente de 1. On sort alors du `while` sans incrémenter `i`. À mesure des appels de `fscanf`, les éléments du tableau sont lus à partir du fichier. À la fin, la variable `i` contient le nombre d'éléments qui ont été lus dans le fichier. La condition du `while` vérifie aussi que l'on ne dépasse pas la taille physique du tableau. Il s'agit d'une sécurité qui évite une erreur de segmentation si le fichier est plus long que le programmeur ne l'a prévu.



Compléments

√ Dans le test du `while` de l'exemple précédent, le test `i < NB_ELEM_MAXI` est effectué avant la lecture par `fscanf`. Ceci est essentiel car la norme *ANSI* spécifie que les conditions dans une conjonction `&&` sont testées dans l'ordre de gauche à droite. Si la première condition est fausse, la deuxième condition n'est pas testée, ce qui évite une erreur mémoire de dépassement de tableau dans le `fscanf`.

Notons enfin qu'à chaque lecture par `fscanf`, le pointeur de fichier passe à la suite dans le fichier. Le pointeur avance automatiquement dans le fichier lorsqu'on effectue une lecture, sans qu'il n'y ait besoin d'incrémenter une variable.

Remarque

Le fichier aurait pu être organisé autrement, en mettant une première ligne contenant le nombre d'éléments à lire et à transférer dans le tableau. La deuxième ligne contientdrait les éléments séparés par des espaces. Par exemple :

```
6

10 2 35 752 -5 4
```

Dans ce cas, on fait un premier `fscanf` pour lire le nombre d'éléments `n` dans le fichier, puis on peut faire une boucle `for` classique, avec comme condition d'arrêt `i<n` pour lire les éléments avec `fscanf`.

La manière dont les données sont organisées dans un fichier s'appelle le *format de fichier*. Le concepteur d'un programme qui utilise des fichiers doit réfléchir et spécifier exactement quelles sont les données dans le fichier et dans quel ordre se trouvent ces données avant d'écrire son programme. Parfois, le format de fichier est

imposé car il peut s'agir d'un standard qui est utilisé par d'autres programmes et logiciels.

10.3.2 Écrire des données formatées

Pour écrire dans un fichier, le fichier doit préalablement avoir été ouvert en mode `"w"`, `"a"`, `"r+"`, `"w+"` ou `"a+"`.

Pour écrire des données numériques (des nombres) ou autre dans un fichier texte, on peut utiliser la fonction `fprintf`, qui est analogue à la fonction `printf`. La fonction `fprintf` prend comme premier paramètre un pointeur de fichier. Les autres paramètres sont les mêmes que pour `printf` ; le second paramètre est la chaîne de format avec le texte à écrire et les `%d,%f`, puis suivent les variables à écrire séparées par des virgules.

> ***Exemple***
>
> Voici un programme qui lit un fichier texte contenant des nombres entiers, et écrit un fichier texte contenant les entiers triples (chaque entier est multiplié par 3).

```
#include <stdio.h>

int TripleFichier(void)
{
   FILE *fpr, *fpw; /* deux pointeurs pour deux fichiers */
   int n;   /* pour lire */

   fpr = fopen("fichierLecture.txt", "rt");
   fpw = fopen("fichierEcriture.txt", "wt");
   if (fpr==NULL || fpw==NULL)  /* gestion d'erreur */
         return 1;  /* code d'erreur retourné au main */
   while (fscanf(fpr, "%d", &n)==1) /* lecture d'un entier */
         fprintf(fpw, "%d ", 3*n);  /* écriture du triple */
   fclose(fpr);  /* fermeture des deux fichiers */
   fclose(fpw);
   return 0;  /* pas d'erreur */
}
int main()
{
   int codeErr;
   codeErr = TripleFichier();
   /* on récupère le code d'erreur dans le main : */
   if(codeErr != 0)
      puts("Erreur d'ouverture de fichier !");
   return 0;
}
```

Compléments

√ La fonction `feof`, qui prend en paramètre le pointeur de fichier, retourne vrai si la fin du fichier a été dépassée. Cela permet de tester la fin du fichier mais cette fonction n'est pas toujours commode car il faut passer l'échec d'une lecture pour que `feof` retourne vrai.

Exercices

10.1 (∗) Une série statistique donne, pour chaque poids : 1*kg*, 2*kg*, 3*kg*,..., 200*kg*, le nombre de personnes de la population qui pèsent ce poids. Ces données sont stockées dans un fichier. La *i*ème ligne du fichier contient le nombre de personnes qui pèsent *i*kg.

```
n1
n2
etc.
n200
```

a) Écrire un programme qui charge ces données en mémoire centrale et calcule le poids moyen dans la population.

b) Écrire un programme qui calcule le poids moyen dans la population sans charger les données en mémoire.

10.2 (∗) Écrire une fonction qui sauvegarde un tableau de `float` passé en paramètre.

10.3 (∗∗) Un fichier contient des descriptions de code article et de prix. Le code d'article est un numéro entre 0 et 99. Chaque ligne du fichier contient un code et un prix séparés par un espace :

```
code1 prix1
code2 prix2
etc.
```

On se propose d'organiser les données en mémoire centrale sous forme de tableaux : on trouve le prix de chaque produit de code *c* dans la case d'indice *c* du tableau.

a) Proposer une structure de données pour représenter les produits en mémoire centrale.

b) Écrire une fonction qui réalise le chargement en mémoire centrale des données.

c) Écrire une fonction qui donne le prix d'un produit à partir de son code.

d) Écrire une fonction qui permet à un utilisateur de rentrer un nouveau produit dans la base. On supposera que la base de données a été préalablement chargée en mémoire.

e) Écrire une fonction qui permet de sauvegarder la base de données dans un fichier.

f) Écrire le programme principal qui charge les données et, tant que l'utilisateur le souhaite, propose un menu avec trois options :

- Ajouter un produit
- Consulter un prix
- Quitter

En fin de programme, la sauvegarde des données sera effectuée.

Corrigés

10.1

```
#define fichier "poids.txt"
#define NB_ELEM_MAXI 200
```

a)

```
int main()
{
  FILE *fp;
  int t[NB_ELEM_MAXI], i, somme;
  fp = fopen(fichier, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture fichier\n");
      return -1;
    }
  i = 0;
  while (fscanf(fp, "%d", &t[i]) == 1)
    i++;
  fclose(fp);
  somme = 0;
  i--;
  while (i >= 0)
```

```
    {
      somme += t[i];
      i--;
    }
  printf("Moyenne = %f\n", somme / (float) NB_ELEM_MAXI);
  return 0;
}
```

b)

```
int main()
{
  FILE *fp;
  int t, n, somme;
  fp = fopen(fichier, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture fichier\n");
      return -1;
    }
  n = 0;
  somme = 0;
  while (fscanf(fp, "%d", &t) == 1)
    {
      somme += t;
      n++;
    }
  fclose(fp);
  printf("Moyenne = %f\n", somme / (float) n);
  return 0;
}
```

10.2

```
#define fichier "float.txt"
void sauveFloat(float t[], int taille)
{
  FILE *fp;
  int i;
  fp = fopen(fichier, "wt");
  if (fp == NULL)
    {
      printf("Erreur de création de fichier\n");
    }
```

```
  else
    {
      for (i = 0; i < taille; i++)
        fprintf(fp, "%f\n", t[i]);
      fclose(fp);
    }
}
```

10.3

```
#define fichier "produit.txt"
#define NB_ELEM_MAXI 100
```

a)

```
typedef struct produit
{
  float prix[NB_ELEM_MAXI];
  /* valide[.] = 1 si le produit existe */
  char valide[NB_ELEM_MAXI];
} Produit;
```

b)

```
Produit chargement()
{
  FILE *fp;
  int i, code;
  float prix;
  Produit base;
  fp = fopen(fichier, "rt");
  for (i = 0; i < NB_ELEM_MAXI; i++)
    base.valide[i] = 0;
  if (fp == NULL)
    {
      printf("Erreur de lecture de fichier\n");
    }
  else
    {
      while (fscanf(fp, "%d %f", &code, &prix) == 2)
        {
          base.valide[code] = 1;
          base.prix[code] = prix;
        }
      fclose(fp);
    }
  return base;
}
```

c)

```
float prix(Produit base, int code)
{
  if (base.valide[code] == 1)
    return base.prix[code];
  /* prix inconnu */
  return -1.0;
}
```

d)

```
Produit ajout(Produit base)
{
  int code;
  float prix;
  printf("Entrez un code et un prix : ");
  scanf("%d %f", &code, &prix);
  base.valide[code] = 1;
  base.prix[code] = prix;
  return base;
}
```

e)

```
void sauve(Produit base)
{
  FILE *fp;
  int i;
  fp = fopen(fichier, "wt");
  if (fp == NULL)
    {
      printf("Erreur de création de fichier\n");
    }
  else
    {
      for (i = 0; i < NB_ELEM_MAXI; i++)
        {
          if (base.valide[i] == 1)
            fprintf(fp, "%d %f\n", i, base.prix[i]);
        }
      fclose(fp);
    }
}
```

f)

```
int main()
{
  int code;
  char c = '0';
  float p;
  Produit base = chargement();
  while (c != '3')
    {
      printf("Voulez-vous :\n");
      printf("1- Ajouter un produit\n");
      printf("2- Consulter un prix\n");
      printf("3- Quitter\n");
      c = '0';
      while (c != '1' && c != '2' && c != '3')
        c = getchar();
      switch (c)
        {
        case '1':
          base = ajout(base);
          break;
        case '2':
          printf("code ? ");
          scanf("%d", &code);
          p = prix(base, code);
          if (p < 0)
            printf("Produit inconnu\n");
          else
            printf("Le prix du produit de code %d est de : %.2f\n",
                   code, prix(base, code));
          break;
        default:
          break;
        }
    }
  sauve(base);
  return 0;
}
```



ADRESSES, POINTEURS ET PASSAGE PAR ADRESSE

11.1 MÉMOIRE CENTRALE ET ADRESSES

La mémoire centrale d'un ordinateur est composée d'un très grand nombre d'octets. Chaque octet est repéré par un numéro appelé *adresse* de l'octet.

adresses	bits							
octet 0								NULL
octet 1								
octet 2								
etc.								
octet $2^n - 2$								
octet $2^n - 1$								

Chaque variable dans la mémoire occupe des octets contigus, c'est-à-dire des octets qui se suivent. Par exemple, un `float` occupe 4 octets qui se suivent. L'*adresse* de la variable est l'adresse de son premier octet. On peut connaître l'adresse d'une variable par l'opérateur `&`.

```
&x  /* adresse de la variable x : adresse de son premier octet */
```

11.2 VARIABLES DE TYPE POINTEUR

L'adresse d'une variable peut elle-même être mémorisée dans une variable. Les variables dont les valeurs sont des adresses s'appellent des *pointeurs*. On déclare un pointeur sur `int` par le type `int*`, un pointeur sur `float` par le type `float*`, etc.

Voici un exemple utilisant un pointeur `p` qui prend pour valeur l'adresse de `x` (on dit que `p` pointe sur `x`, voir la figure 11.1).

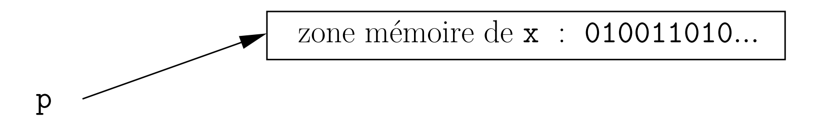


Figure 11.1– Le pointeur p pointe sur la variable x

```
#include <stdio.h>

int main(void)
{
   int x = 2; /* déclaration d'une variable x */
   int *p; /* déclaration d'un pointeur p */
   p = &x; /* p pointe sur x */
   /* la valeur de p est l'adresse de x */
   scanf("%d", p);  /* lecture de la valeur de x au clavier */
   printf("%d", x);  /* affichage de la nouvelle valeur de x */
   return 0;
}
```

On accède à la donnée pointée par un pointeur (valeur de x dans l'exemple précédent) par une étoile.

```
*p = 3;    /* l'objet pointé par p prend pour valeur 3 */
```



Ne pas confondre l'usage de l'étoile lors de la déclaration d'une variable de type pointeur avec l'usage de l'étoile qui permet d'accéder à l'objet pointé par le pointeur.



Ne pas confondre la valeur d'un pointeur p, qui est une adresse, et la valeur de l'objet pointé par p, qui n'est en général pas une adresse, par exemple un int dans le cas d'un pointeur de type int*.

```
#include <stdio.h>

int main(void)
{
   int x = 2;
   int *p = &x;  /* x et *p deviennent synonymes */
   *p = 3;
   printf("La nouvelle valeur de x est %d\n", x);
   /* doit afficher la valeur 3 */
   return 0;
}
```

```
#include <stdio.h>

int main(void)
{
   int x = 2;
   int *p = &x;  /* x et *p deviennent synonymes */
   printf("La valeur de x est %d\n", *p); /* affiche 2 */
   x = 5;
   printf("La nouvelle valeur de x est %d\n", *p);
   /* doit afficher la valeur 5 */
   return 0;
}
```

En résumé, lorsque p pointe sur x, la valeur de p est l'adresse de x, toute modification de *p modifie x et toute modification de x modifie *p. La raison est que *p et x sont sur le même emplacement mémoire dans la mémoire RAM.

11.3 Passage de paramètre par valeur

Lorsqu'on passe un paramètre à une fonction, la fonction ne peut pas modifier la variable. La variable est automatiquement recopiée et la fonction travaille sur une copie de la variable. La modification de la copie n'entraîne pas une modification de la variable originale. C'est le passage de paramètre par valeur.

```
#include <stdio.h>

void NeModifiePas(int x)
{
   x = x+1;  /* le x local est modifié, pas le x du main */
}

int main(void)
{
   int x=1;
   NeModifiePas(x);
   printf("%d", x); /* affiche 1 : valeur de x inchangée */
   return 0;
}
```

11.4 Passage de paramètre par adresse

L'idée du passage par adresse est que, pour modifier une variable par un appel de fonction, il faut passer en paramètre non pas la variable, mais un pointeur qui pointe

sur la variable. Ainsi, le paramètre est l'adresse de x. Lorsqu'on modifie la mémoire à cette adresse, la donnée x est modifiée, car on travaille bien sur l'emplacement mémoire de x (voir figure 11.2).

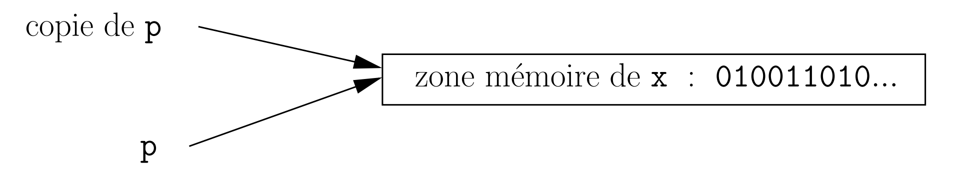


Figure 11.2 – Le pointeur p et sa copie pointent tous deux sur la variable x du main

```
#include <stdio.h>

/* la fonction suivante prend en paramètre un pointeur */
void Modifie(int *p)
{
   *p = *p+1;  /* p pointe sur x, la copie de p aussi */
               /* le x du main est modifié */
}

int main(void)
{
   int x=1;  /* la varible x n'est pas un pointeur */
   int *p;
   p = &x;  /* pointeur qui pointe sur x */
   Modifie(p);
   printf("%d", x);  /* affiche 2 */
   return 0;
}
```

Lors de l'appel de la fonction Modifie, le pointeur p est recopié, et la fonction Modifie travaille sur une copie du pointeur p. Cependant, les pointeurs p et sa copie contiennent la même valeur, qui est l'adresse de x. Lorsqu'on modifie l'objet qui se trouve à cette adresse, on modifie x.

L'utilisation explicite d'un pointeur dans le main est superflue, on peut passer directement l'adresse de x à la fonction, sans avoir besoin de définir une variable de type pointeur dans le main.

```
#include <stdio.h>

void Modifie(int *p)  /* paramètre de type pointeur */
{
   *p = *p+1; /* ce pointeur p a pour valeur  &x (x du main) */
```

```
}

int main(void)
{
   int x=1;
   Modifie(&x); /* on passe directement l'adresse de x */
   printf("%d", x);  /* affiche 2 */
   return 0;
}
```

C'est sous cette dernière forme que l'on utilise en général le passage par adresse.

Exemple

La fonction `scanf`, qui lit des variables au clavier, doit modifier le contenu de ces variables. Pour cette raison, les variables sont passées à `scanf` par adresse, ce qui explique la nécessité des `&` devant les variables.



Avant d'utiliser l'objet pointé par un pointeur `p`, on doit s'assurer que le pointeur `p` contient l'adresse d'un emplacement mémoire correct (adresse d'une variable ou d'un bloc mémoire alloué dynamiquement comme nous l'étudierons au chapitre suivant). Si l'on déclare un pointeur `p` sans le faire pointer sur un emplacement mémoire correct, tout accès à `*p` produira soit un résultat faux, soit une erreur mémoire (erreur de segmentation).

Exercices

11.1 (∗) Écrire une fonction qui initialise deux entiers et un réel à 0. Écrire le programme principal qui appelle cette fonction.

11.2 (∗) Écrire une fonction qui retourne le quotient et le reste de la division d'un entier p par un entier q. Écrire le programme principal qui appelle cette fonction et affiche les résultats.

11.3 (∗∗)

a) Écrire un programme qui saisit deux variables de type `int`, qui échange leur contenu et qui affiche les nouvelles valeurs des variables.

b) On se propose de refaire la question a) en réalisant l'échange de valeurs à l'intérieur d'une fonction. Écrire une fonction qui échange le contenu de deux variables passées par adresse. Écrire le programme principal qui saisit deux variables, les échange en appelant la fonction et affiche le nouveau contenu des variables.

11.4 (∗) Écrire une fonction qui initialise toutes les valeurs d'un tableau à 0. Écrire le programme principal qui déclare le tableau, appelle la fonction et affiche les éléments du tableau.

11.5 (∗) Écrire une fonction qui calcule la somme et le produit des éléments d'un tableau passé en paramètre. Écrire le programme principal qui initialise le tableau par saisie ; calcule et affiche la somme et le produit des éléments.



Compléments

√ Pour l'exercice suivant, on doit utiliser des pointeurs sur une structure. Si `p` est un pointeur sur une structure contenant un champ `x` de type `float`, alors `(*p)` est une structure, et `(*p).x` est un `float`. Pour alléger l'écriture, on peut utiliser la notation `p->x` à la place de `(*p).x` pour désigner le champ `x` de la structure pointée par `p`.

11.6 (∗∗) Soit une structure `Point` contenant deux champs *x* et *y* de type `float`.

a) Écrire une fonction qui échange deux structures `Point` passées par adresse.

b) Écrire le programme principal qui saisit deux structures `Point` dans des variables, échange le contenu de ces variables en appelant la fonction et affiche le nouveau contenu des variables.

Corrigés

11.1

```
void remplir(int *a, int *b, float *x)
{
  *a = 0;
  *b = 0;
  *x = 0.0;
}
int main()
{
  int u, v;
  float t;
  remplir(&u, &v, &t);
  return 0;
}
```

11.2

```
void divise(int p, int q, int *quotient, int *reste)
{
  *reste = p % q;
  *quotient = p / q;
}
int main()
{
  int a, b;
  divise(17, 3, &a, &b);
  printf("17 = %d * 3 + %d\n", a, b);
  return 0;
}
```

11.3

a)

```
int main()
{
  int a = 14, b = 5;
  int t;
  printf("a = %d; b = %d\n", a, b);
  t = a;
  a = b;
  b = t;
  printf("a = %d; b = %d\n", a, b);
  return 0;
}
```

b)

```
void echange(int *a, int *b)
{
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
int main()
{
  int a = 14, b = 5;
  printf("a = %d; b = %d\n", a, b);
  echange(&a, &b);
  printf("a = %d; b = %d\n", a, b);
  return 0;
}
```

11.4

```
#define NB_ELEM_MAXI 100
void raz(int *t, int taille)
{
  int i;
  for (i = 0; i < taille; i++)
    t[i] = 0;
}
int main()
{
  int tab[NB_ELEM_MAXI];
  int i;
  for (i = 0; i < NB_ELEM_MAXI; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
  raz(tab, NB_ELEM_MAXI);
  for (i = 0; i < NB_ELEM_MAXI; i++)
    printf("tab[%d] = %d\n", i, tab[i]);
  return 0;
}
```

11.5

```
#define NB_ELEM_MAXI 100
void calcul(int t[], int taille, int *somme, int *produit)
{
  int i;
  *somme = 0;
  *produit = 1;
  for (i = 0; i < taille; i++)
    {
      *somme += t[i];
      *produit *= t[i];
    }
}
int main()
{
  int tab[NB_ELEM_MAXI];
  int s, p, taille=-1, i;
  while (taille < 0 || taille > NB_ELEM_MAXI)
    {
      printf("Entrez le nombre d’éléments : ");
      scanf("%d", &taille);
    }
  puts("Veuillez saisir les éléments du tableau");
```

```
  for (i=0 ; i<taille ; i++)
    scanf("%d", &tab[i]);
  calcul(tab, taille, &s, &p);
  printf("somme = %d et produit = %d\n", s, p);
  return 0;
}
```

11.6

```
typedef struct point
{
  float x, y;
} Point;
```

a)

```
void echange(Point * a, Point * b)
{
  Point t;
  t.x = a->x;
  t.y = a->y;
  a->x = b->x;
  a->y = b->y;
  b->x = t.x;
  b->y = t.y;
}
```

b)

```
int main()
{
  Point M, N;
  printf("Coordonnées du premier point ? \n");
  scanf("%f %f", &M.x, &M.y);
  printf("Coordonnées du second point ? \n");
  scanf("%f %f", &N.x, &N.y);
  echange(&M, &N);
  printf("M(%f,%f) et N(%f,%f)\n", M.x, M.y, N.x, N.y);
  return 0;
}
```

Allocation dynamique 12

12.1 Gestion de la mémoire centrale

Jusqu'à maintenant, le nombre des éléments d'un tableau était limité par une constante (en général définie dans un `#define`). Dans cette partie, nous verrons comment créer un tableau en cours de programme, la taille de ce tableau pouvant être donnée par une variable, et résulter d'un calcul ou être saisie par l'utilisateur. L'outil pour faire cela est l'*allocation dynamique* de mémoire. Cette technique permet de créer des tableaux dont la taille mémoire est variable en fonction des besoins, et de libérer cette mémoire après utilisation. On obtient ainsi des programmes plus performants en termes de consommation de mémoire.

En général, plusieurs programmes ou logiciels s'exécutent en même temps sur un même ordinateur. Par exemple, on peut faire tourner simultanément une console, un éditeur de texte, un traitement de texte, un compilateur, etc. On parle de système multitâche. Tous ces programmes ont chacuns leurs données en mémoire (variables, tableaux...), mais les programmes se partagent en général le même matériel, et en particulier les mêmes barettes mémoires. Le système doit allouer la mémoire aux différents programmes, sans qu'il y ait de conflits. En particulier, un programme ne peut pas écrire dans les variables d'un autre programme ; cela provoque une erreur mémoire, ou *erreur de segmentation*.

Pour créer un tableau en cours d'exécution du programme, il faut réserver un emplacement dans la mémoire centrale. Le programmeur indique la taille de l'emplacement mémoire (en gros le nombre d'octets), et, lors de l'exécution du programme, le système réserve un emplacement mémoire et donne l'adresse de cet emplacement. Cette opération s'appelle l'*allocation dynamique* de mémoire. Les fonctions d'allocation en *C* sont les fonctions *malloc* et *calloc*. On parle d'allocation dynamique parce que l'allocation de mémoire a lieu à mesure des besoins, par opposition à l'allocation statique de tableaux étudiée au chapitre 9.

12.2 Allocation avec `malloc`

La fonction `malloc` alloue un certain nombre d'octets, c'est-à-dire qu'elle réserve des octets pour une utilisation par le programme. Le nombre d'octets est passé en paramètre à la fonction `malloc`. La fonction `malloc` retourne l'adresse du premier octet réservé. On mémorise cette adresse dans un pointeur. Pour calculer le nombre total d'octets nécessaires à un tableau, on utilise la fonction `sizeof` qui donne le nombre

d'octets nécessaires à une variable de type `float`, `int`, `char`, etc. et on multiplie par le nombre d'éléments du tableau.

Après utilisation de la mémoire, la mémoire doit **impérativement** être libérée avec la fonction `free`. Celle-ci prend en paramètre l'adresse du bloc mémoire qui doit être libéré. Lors de l'appel à `free`, la mémoire est rendue au système qui peut la réemployer pour une autre utilisation.

Exemple

Dans le programme suivant, la fonction `RentrerTableau` lit le nombre d'éléments d'un tableau, réserve de l'espace mémoire pour des `float` en conséquence, lit les éléments du tableau au clavier et retourne l'adresse du tableau. Le nombre d'éléments du tableau est passé par adresse pour être transmis au `main`. Notons que le nombre d'élements du tableau est ici donné par une variable, et non pas une constante.

```
#include <stdio.h>
#include <stdlib.h>    /* pour utiliser malloc */

/* fonction retournant un pointeur de type float* */
float* RentrerTableau(int *addrNbreElements)
{
   int n, i;
   float *tab;    /* adresse du tableau (type pointeur) */
   printf("Entrez la taille du tableau : ");
   scanf("%d", &n);
   *addrNbreElements = n; /* passage par adresse, renvoi de n */
   /* le nombre d'éléments connu, on alloue le tableau */
   tab = (float*)malloc(n*sizeof(float));  /* allocation */
   puts("Entrez les éléments du tableau :");
   for (i=0 ; i<n ; i++)
      scanf("%f", &tab[i]);
   return tab;    /* on retourne l'adresse du tableau */
}

void Affichage(float *tab, int nb) /* affiche un tableau tab */
{
   int i;
   for (i=0 ; i<nb ; i++)
      printf("tab[%d] = %f\n", i, tab[i]);
}

int main(void)
{
   int nb;
```

```
    float *tab;
    tab = RentrerTableau(&nb);
    /* on récupère l'adresse du tableau dans tab */
    Affichage(tab, nb);
    free(tab)  /* libération de mémoire obligatoire */
    return 0;
}
```

La fonction `malloc`, qui alloue la mémoire, retourne l'adresse de l'emplacement mémoire alloué. On peut mémoriser cette adresse dans une variable pouvant contenir des adresses : une variable de type pointeur. Dans cette représentation, un tableau est toujours donné par une variable de type pointeur qui contient l'adresse du premier élément du tableau. On accède aux éléments du tableau comme pour n'importe quel tableau : par les crochets `[ ]`.

Notons que, contrairement aux tableaux statiques, que l'on passe en paramètre aux fonctions pour modifier les éléments, on peut retourner un tableau alloué dynamiquement par un `return`. La fonction retourne alors simplement la valeur d'un pointeur (type `float*` dans l'exemple).

Notons le passage par adresse qui permet à la fonction `RentrerTableau` de transmettre au `main` à la fois le tableau et son nombre d'éléments. Une alternative serait de mettre le tableau (en tant que pointeur) et son nombre d'éléments dans une même structure, et de retourner la structure.

12.3 Allocation avec calloc

La fonction `calloc`, comme la fonction `malloc`, alloue un emplacement mémoire et retourne l'adresse du premier octet. La syntaxe des deux fonctions diffère légèrement. La fonction `calloc` prend deux paramètres, le nombre d'éléments et le nombre d'octets de chaque élément. La fonction `calloc`, contrairement à `malloc`, initialise tous les octets à 0.

Exemple

Dans l'exemple suivant, on alloue un tableau dont tous les éléments sont nuls sauf ceux rentrés par l'utilisateur.

```
#include <stdio.h>
#include <stdlib.h>   /* pour utiliser calloc */

float* RentrerTableauBis(int *addrNbreElements)
{
    int i, n;
    char choix;
```

```
   float *tab;    /* adresse du tableau */
   printf("Entrez la taille du tableau : ");
   scanf("%d", &n);
   *addrNbreElements = n;  /* passage par adresse */
   /* le nombre d'éléments connu, on alloue le tableau */
   tab = (float*)calloc(n, sizeof(float));  /* allocation */
   puts("Entrez les éléments non nuls du tableau :");
   choix = 'y';
   while (choix=='y')
   {
      puts("Entrez l'indice i de l'élément à saisir");
      scanf("%d", &i);
      puts("Entrez l'élément tab[i]");
      scanf("%f", &tab[i]);
      getchar(); /* pour manger le retour chariot */
      puts("Y-a-t'il un autre élément non nul ? (y/n)");
      choix = getchar();
   }
   return tab;    /* on retourne l'adresse du tableau */
}

void Affichage(float *tab, int nb)
{
   int i;
   for (i=0 ; i<nb ; i++)
      printf("tab[%d] = %f\n", i, tab[i]);
}

int main(void)
{
   int nb;
   float *tab;
   tab = RentrerTableauBis(&nb);
   Affichage(tab, nb);
   free(tab)  /* libération de mémoire obligatoire */
   return 0;
}
```

Compléments

√ Les tableaux statiques, comme les tableaux dynamiques, sont représentés par l'adresse de leur premier élément. Lorsqu'on déclare un tableau statique dans une fonction, la mémoire est locale à cette fonction et est détruite lorsque le programme sort de la fonction. En particulier, on ne peut pas retourner un tel tableau (la mémoire est détruite). En revanche, la mémoire d'un tableau alloué

dynamiquement subsiste jusqu'à ce qu'elle soit libérée "manuellement" par un appel à `free`.

√ Les deux types de mémoire, statique et dynamique, ne sont pas stockés dans la même zone de la *RAM*. La mémoire statique, ainsi que les variables des fonctions, est stockée dans une pile (appelée pile d'appels). La mémoire dynamique est stockée dans un tas. La pile et le tas sont gérés de manière très différente par le système. En général dans les paramètres du système par défaut, la pile est de petite taille et le tas permet d'allouer beaucoup plus de mémoire.

Exercices

12.1 (∗) Écrire une fonction qui lit un entier n au clavier, alloue un tableau de n entiers initialisés à 0, et retourne n et le tableau.

12.2 (∗) On se propose de réaliser une fonction de chargement en mémoire centrale sous forme de tableau de `float` d'un fichier texte. Le format de fichier est le suivant :

- la première ligne du fichier contient le nombre d'éléments du tableau ;
- les lignes suivantes contiennent chacune un nombre réel.

```
n
f_1
f_2
...
f_n
```

a) Réaliser une fonction de chargement dans un tableau dont la taille mémoire correspond exactement au nombre d'éléments du fichier.

b) Réaliser une fonction d'affichage du tableau.

c) Écrire le programme principal qui charge le fichier et affiche le tableau.

12.3 (∗) Soit la suite u_n définie par :

$$u_0 = 1 \text{ et } u_{n+1} = 3u_n^2 + 2u_n + 1$$

a) Écrire une fonction qui prend en paramètre un entier n et qui retourne un tableau contenant les n premiers termes de la suite u_n. La fonction doit marcher quel que soit l'entier n rentré.

b) Écrire le programme principal qui saisit l'entier n et affiche les n premiers termes de la suite u_n en utilisant la fonction définie au a).

12.4 (∗∗) Considérons la structure `TypeTableau` suivante qui contient l'adresse d'un tableau et le nombre d'éléments du tableau.

```
typedef struct{
                 int nb_elem;  /* nombre d'éléments */
                 int *tab;     /* tableau */
                }TypeTableau;
```

a) Écrire une fonction de prototype

`TypeTableau CreationTableau(int n);`

qui crée un tableau de `n` éléments.

b) Écrire une fonction de prototype

`void DestructionTableau(TypeTableau T);`

qui libère la mémoire occupée par un tableau.

c) Écrire une fonction de prototype

`void SimpleLectureTableau(TypeTableau T);`

qui lit les éléments d'un tableau au clavier. On supposera dans cette fonction que le tableau a déja été alloué précédemment.

d) Écrire une fonction de prototype

`void Affichage(TypeTableau T);`

qui affiche le contenu d'un tableau.

e) Écrire une fonction de prototype

`TypeTableau DoubleTableau(TypeTableau T);`

qui crée un nouveau tableau de même taille que `T` mais dont les éléments sont le double des éléments de `T`.

f) Écrire un programme principal qui saisit un tableau au clavier, calcule le double de chaque élément du tableau et affiche les résultats.

Corrigés

12.1

```
int *zero(int *n)
{
  int *t = NULL;
  printf("Entrez une taille : ");
  scanf("%d", n);
  if (*n > 0)
    {
      t = calloc(*n, sizeof(int));
    }
  return t;
}
```

12.2

```
#define fichier "donnees.dat"
```

a)

```
float *charge(FILE * fpr, int *n)
{
  float *t = NULL;
  int i = 0;
  fscanf(fpr, "%d", n);
  if (*n > 0)
    {
      t = calloc(*n, sizeof(float));
      while (fscanf(fpr, "%f", &t[i]) == 1)
        i++;
    }
  return t;
}
```

b)

```
void affiche(float *t, int n)
{
  int i;
  for (i = 0; i < n; i++)
    printf("t[%d] = %f\n", i, t[i]);
}
```

c)

```
int main()
{
  float *t;
  int n;
  FILE *fp;
  fp = fopen(fichier, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture\n");
      return -1;
    }
  t = charge(fp, &n);
  affiche(t, n);
  return 0;
}
```

12.3

a)

```
unsigned int *suite(unsigned int n)
{
  unsigned int *un = NULL;
  unsigned int i;
  if (n <= 0)
    return un;
  un = calloc(n, sizeof(unsigned int));
  if (un != NULL)
    {
      un[0] = 1;
      i = 1;
      while (i < n)
        {
          un[i] = 3 * un[i - 1] * un[i - 1] + 2 * un[i] + 1;
          i++;
        }
    }
  return un;
}
```

b)

```
int main()
{
  unsigned int n;
  unsigned int *u;
```

```
  unsigned int i;
  printf("Nombre de termes ? ");
  scanf("%u", &n);
  u = suite(n);
  if (u == NULL)
    return -1;
  i = 0;
  while (i < n)
    {
      printf("u_%u = %u\n", i, u[i]);
      i++;
    }
  return 0;
}
```

12.4

```
typedef struct
{
  int nb_elem;
  int *tab;
} TypeTableau;
#define N 5
```

a)

```
TypeTableau CreationTableau(int n)
{
  TypeTableau t;
  t.nb_elem = 0;
  t.tab = NULL;
  if (n > 0)
    {
      t.tab = calloc(n, sizeof(int));
      if (t.tab != NULL)
        t.nb_elem = n;
    }
  return t;
}
```

b)

```
void DestructionTableau(TypeTableau T)
{
  if (T.nb_elem > 0)
    {
      free(T.tab);
      T.nb_elem=0;
    }
}
```

c)

```
void SimpleLectureTableau(TypeTableau T)
{
  int i;
  for (i = 0; i < T.nb_elem; i++)
    {
      printf("T[%d] = ? ", i);
      scanf("%d", &(T.tab[i]));
    }
}
```

d)

```
void Affichage(TypeTableau T)
{
  int i;
  for (i = 0; i < T.nb_elem; i++)
    {
      printf("T[%d] = %d\n", i, T.tab[i]);
    }
}
```

e)

```
TypeTableau DoubleTableau(TypeTableau T)
{
  TypeTableau TT = CreationTableau(T.nb_elem);
  int i;
  for (i = 0; i < T.nb_elem; i++)
    {
      TT.tab[i] = 2 * T.tab[i];
    }
  return TT;
}
```

f)

```
int main()
{
  TypeTableau Ta, TaTa;
  Ta = CreationTableau(N);
  SimpleLectureTableau(Ta);
  TaTa = DoubleTableau(Ta);
  Affichage(TaTa);
  DestructionTableau(Ta);
  DestructionTableau(TaTa);
  return 0;
}
```

CHAÎNES DE CARACTÈRES



13.1 QU'EST-CE QU'UNE CHAÎNE DE CARACTÈRES ?

Une chaîne de caractères est un tableau de caractères se terminant par le caractère spécial `'\0'` (qui a 0 pour code *ASCII*).

'e'	'x'	'e'	'm'	'p'	'l'	'e'	'\0'

Le caractère `'\0'` sert à repérer la fin de la chaîne, évitant d'avoir à connaître le nombre de caractères de la chaîne. On peut ainsi passer une chaîne de caractères en paramètre à une fonction, sans avoir besoin de passer un deuxième paramètre contenant le nombre de catactères.

On peut déclarer une chaîne de caractères comme n'importe quel tableau, mais en prévoyant une place pour le `'\0'` final :

```
/* ******* version allocation statique ******** */

char chaine[100];

/* ****** version allocation dynamique ******** */

char *chaine;  /* ne pas oublier l'allocation */
int n = 100;   /* variable n */
chaine = (char*)calloc(n, sizeof(char));  /* allocation */
```

Exemple

Le programme suivant montre une fonction qui calcule la longueur d'une chaîne, c'est-à-dire son nombre de caractères sans compter le `'\0'`.

```
#include <stdio.h>

int longueur(char* chaine)
{
   int i;
   /* on teste la fin de chaîne avec le '\0'*/
   for (i=0 ; chaine[i] != '\0' ; i++)
```

```
      {}  /* bloc d'instructions vide */
   return i;
}
int main(void)
{
   char chaine[101];
   int long;
   puts("Veuillez entrer une chaîne (au plus 100 caractères)");
   puts(" sans espaces"); /* scanf s'arrête au premier espace */
   scanf("%s", chaine);   /* lecture de la chaîne. pas de & */
   long = longueur(chaine);
   printf("Longueur de la chaîne = %d", long);
}
```



On ne met pas de `&` dans `scanf` pour lire une chaîne de caractères. Ceci est dû au fait qu'un chaîne de caractères est déjà une adresse, et qu'il n'y a pas besoin dans `scanf` de passer l'adresse du pointeur contenant l'adresse du bloc mémoire.

Il existe aussi des constantes de type chaîne de caractères. Celles-ci son composées de caractères entre guillements.

```
#define MA_CHAINE "voici une constante de type chaîne"
...
puts(MA_CHAINE);   /* affichage de la chaîne */
```



Ne pas mélanger les constantes de type chaîne (type `char*`), qui sont entre guillemets (ou doubles quotes), et les constantes de type caractère (type `char`) qui sont entre simples quotes comme `'Z'`.

13.2 Opérations prédéfinies sur les chaînes

13.2.1 Fonctions de `<stdio.h>`

a) Lecture

Le format %s de `scanf` et `fscanf` permet de lire une chaîne de caractères au clavier ou dans un fichier texte. La chaîne se termine dès qu'on rencontre un espace ou un `'\n'`.

Exemple 1

```
#include <stdio.h>
int main(void)
{
   char chaine[100];
   puts("Veuillez entrer un mot");
```

```
    scanf("%s", chaine);
    printf("Vous avez entré %s", chaine);
    return 0;
}
```

La fonction `fgets` lit toute une ligne dans un fichier texte. La chaîne lue **peut contenir des espaces**. La chaîne lue se termine par un `'\n'`, qui doit être supprimé “à la main” s'il est indésirable. La fonction `fgets` a pour prototype :

```
char *fgets(char* s, int n, FILE *fp);
```

La fonction lit tous les caractères jusqu'au prochain `'\n'`, mais elle lit au plus $n-1$ caractères. Les caractères lus sont mis dans `s`, **y compris le** `'\n'`. Le paramètre `n` sert à éviter une erreur mémoire si la ligne lue est trop longue pour le tableau alloué. Si aucun `'\n'` n'est rencontré, la fonction met un `'\0'` sans lire la suite. Il faut avoir préalablement alloué (statiquement ou dynamiquement) la chaîne `s` avec au moins `n` caractères. La fonction `fgets` retourne la chaîne `s`.

Si on passe le pointeur de fichier standard `stdin` comme troisième paramètre, la lecture se fait au clavier et non dans un fichier.



Compléments

√ Toutes les fonctions de lecture ou d'écriture de texte dans des fichiers peuvent être utilisées pour lire ou écrire au clavier. Il suffit pour cela d'utiliser comme pointeur de fichier (de type `FILE*`) le flot d'entrée standard `stdin` et de sortie standard `stdout`. Ceci nous laisse entrevoir le fait que le type `FILE*` s'applique à bien d'autres choses que des fichiers. C'est la notion de *flot*, qui permet de gérer toutes les entrées sorties en *C*.

Exemple 2

```
#include <stdio.h>
int main(void)
{
    char chaine[100];
    puts("Veuillez entrer une ligne, puis appuyez sur entrée");
    fgets(chaine, 100, stdin);
    printf("Vous avez saisi %s", chaine);
    return 0;
}
```

Exemple 3

```
#include <stdio.h>
int main(void)
{
   char chaine[500], nomfich[100];
   FILE *fp;
   puts("Veuillez entrer un nom de fichier sans espace");
   scanf("%s", nomfich);
   if ((fp = fopen(nomfich, "rt")) == NULL)
      puts("Erreur : fichier inexistant ou droits insuffisants");
   else
      {
          fgets(chaine, 500, fp);  /* lecture d'une ligne */
          printf("La première ligne du fichier est %s", chaine);
      }
   return 0;
}
```

b) Écriture

Le format `%s` de `printf` et `fprintf` permet d'écrire une chaîne de caractères à l'écran ou dans un fichier texte. (Voir l'exemple 1.)

La fonction `puts` permet d'afficher une chaîne de caractères suivie d'un `'\n'`

Exemple 4

```
#include <stdio.h>

int main(void)
{
   char chaine[100];
   puts("Veuillez saisir un mot");
   scanf("%s", chaine);
   printf("Vous avez saisi : ")
   puts(chaine); /* pas de guillemets : variable */
   return 0;
}
```



On ne met pas de guillemets dans la fonction `puts` pour afficher le contenu d'une variable de type `char*`. Une instruction `puts("chaine")` afficherait le mot “chaine”, et non pas le contenu de la variable `chaine` qui serait un autre mot ou une phrase.

La fonction `fputs` permet d'écrire une chaîne de caractères dans un fichier texte. Cette fonction a pour prototype :

```
char *fputs(char* s, FILE* fp);
```

La fonction retourne `EOF` en cas d'erreur.

13.2.2 La bibliothèque `<string.h>`

La bibliothèque `string.h` contient des fonctions de traitement des chaînes de caractères. Nous ne donnons ici que quelques exemples de fonctions de `string.h`.

La fonction `strcpy` copie une chaîne dans une autre. C'est l'équivalent d'une affectation pour les chaînes de caractères. Le prototype de la fonction `strcpy` est le suivant :

```
char* strcpy(char* destin, char*source);
```

La fonction copie la chaîne `source` dans `destin`. La chaîne `destin` doit avoir été préalablement allouée. La fonction retourne `destin`.



Lorsqu'on fait une affectation :

```
char  s1[50], *s2;

strcpy(s1, "Ceci est une chaîne");

s2 = s1;  /* affectation */
```

la chaine `s1` n'est pas recopiée et les deux chaînes `s1` et `s2` pointent sur la même zone mémoire (affectation d'adresse). Par la suite, toute modification des caractères de `s1` entraîne une modification de `s2` et réciproquement. Pour recopier la chaîne `s1`, il faut utiliser `strcpy(s2,s1)`.

La fonction `strcat` concatène deux chaînes de caractères, c'est-à-dire qu'elle les met bout à bout l'une à la suite de l'autre dans une même chaîne. Le prototype est :

```
char* strcat(char* s1, char* s2);
```

La fonction recopie `s2` à la suite de `s1` Le résultat se trouve dans `s1`. La chaîne `s1` doit avoir été allouée avec suffisament de mémoire pour contenir le résultat. La fonction retourne `s1`.

Exemple 5

```
#include <stdio.h>
#include <string.h>

int main(void)
{
   char chaine[100], mot1[50], mot2[51];
   puts("Veuillez saisir un mot (au plus 49 lettres)");
   scanf("%s", mot1);
   puts("Veuillez saisir un autre mot (au plus 50 lettres)");
```

```
   scanf("%s", mot2);
   strcpy(chaine, mot1);  /* on copie mot1 dans chaine */
   strcat(chaine, mot2);  /* on met mot2 à la suite */
   printf("Les deux mots saisis sont %s", chaine);
   return 0;
}
```

Le fonction `strcmp` permet de comparer deux chaînes pour l'ordre alphabétique. Le prototype est :

```
int strcmp(char* s1, char *s2);
```

Le résultat est < 0 si $s1 < s2$, égal à 0 si les caractères de $s1$ sont les mêmes que les caractères de $s2$, et il est > 0 si $s1 > s2$.



Le test `cs==ct` est une comparaison des adresses et non pas une comparaison alphabétique.

Exemple 6

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char mot1[50], mot2[50];
   puts("Veuillez saisir un mot (au plus 49 lettres)");
   scanf("%s", mot1);
   puts("Veuillez saisir autre mot (au plus 49 lettres)");
   scanf("%s", mot2);
   if (strcmp(mot1, mot2)==0)
      puts("les deux mots sont égaux");
   if (strcmp(mot1, mot2) < 0)
      printf("%s vient avant %s dans l'ordre alphabétique",
             mot1, mot2);
   if (strcmp(mot1, mot2) > 0)
      printf("%s vient après %s dans l'ordre alphabétique",
             mot1, mot2);
   return 0;
}
```

La fonction `strlen` retourne la longueur d'une chaîne de caractères, c'est-à-dire son nombre de caractères (sans compter le `'\0'`). Cette fonction prend en paramètre la chaîne de caractères.

```
size_t strlen(char* s);
```

Le type `size_t` est un type entier qui sert à stocker des nombres d'octets.

Exemple 7

La fonction suivante permet de lire une ligne au clavier et supprime le '\n' à la fin de la chaîne.

```
void SaisieLigne(char chaine[502])
{
   puts("Veuillez entrer une ligne d'au plus 500 caractères");
   fgets(chaine, 502, stdin);
   /* On remplace le '\n' (dernier caractère) par un '\0'*/
   chaine[strlen(chaine)-1] = '\0';
}
```

Exemple 8

La fonction suivante retourne une chaîne allouée avec juste assez de place en mémoire pour la chaîne saisie au clavier.

```
/* Allocation d'une chaîne juste de la bonne taille */

char* SaisieChaine(void)
{
   char chaine[1000];
   char *econome;
   puts("Veuillez entrer une ligne (au plus 999 lettres)");
   fgets(chaine, 1000, stdin);
   econome = (char *)calloc((strlen(chaine)+1)*sizeof(char));
   strcpy(econome, chaine);
   return econome;
}
```

Exercices

Sans utiliser `string.h`

13.1 (∗) Faire une fonction qui prend en paramètre une chaîne, et renvoie le nombre d'occurences de la lettre f dans la chaîne.

13.2 (∗) Faire une fonction qui renvoie la somme des codes $ASCII$ des caractères d'une chaîne.

13.3 (∗) Faire une fonction qui recopie une chaîne et renvoie une copie de cette chaîne.

13.4 (∗∗) Faire une fonction qui renvoie la concaténation de deux chaînes, c'est-à-dire une chaîne constituée des deux chaînes mises bout à bout.

13.5 (∗) Écrire une fonction de comparaison qui prend en paramètre deux chaînes, renvoie : -1 si la première chaîne est inférieure à la deuxième dans l'ordre alphabétique ; 0 si les deux chaînes sont égales ; +1 si la première chaîne est supérieure à la deuxième dans l'ordre alphabétique.

On conviendra que l'ordre alphabétique revient à la comparaison des codes $ASCII$ des caractères.

13.6 (∗) Écrire une fonction qui lit une chaîne de caractères au clavier et compte le nombre d'espaces contenus dans cette chaîne.

En utilisant éventuellement `string.h`

13.7 (∗) **a**) Écrire une fonction qui lit un nom de fichier au clavier, qui ouvre un fichier de ce nom et compte le nombre d'occurences d'un mot dans le fichier. On supposera que le mot ne contient pas d'espace et que le fichier ne contient pas de ponctuation. Le mot doit être passé en paramètre.

b) Écrire le programme principal qui lit un mot au clavier et appelle la fonction.

13.8 (∗) Écrire une fonction qui ouvre un fichier texte dont le nom est passé en paramètre et qui calcule la somme des longueurs de ses lignes.

13.9 (∗) Écrire une fonction qui prend en paramètre le nom d'un fichier texte et détermine le nombre de lignes du fichier qui commencent par le mot “programmons”.

Corrigés

13.1

```
int occ(char *s)
{
  int occurrence = 0;
  int t = 0;

  while (s[t] != 0)
    {
      if (s[t] == 'f')
        occurrence++;
      t++;
    }
  return occurrence;
}
```

13.2

```
int ascii(char *s)
{
  int somme = 0;
  int t = 0;
  while (s[t] != 0)
    {
      somme += s[t];
      t++;
    }
  return somme;
}
```

13.3

```
char *recopie(char *s)
{
  int taille = 0;
  int t,u;
  char *copie;
  while (s[taille] != '\0')
    {
      s[taille]++;
    }
  copie = calloc(taille + 1, sizeof(char));
```

```
  if (copie != NULL)
    {
      t = 0;
      u = 0;
      while (s[t] != '\0')
        {
          copie[u] = s[t];
          t++;
          u++;
        }
      copie[u] = '\0';
    }
  return copie;
}
```

13.4

```
char *concat(char *s, char *t)
{
  int tailles = 0, taillet = 0;
  int tmps, tmpt, tmpc;
  char *concat;
  while (s[tailles] != '\0')
    {
      tailles++;
    }
  while (t[taillet] != '\0')
    {
      taillet++;
    }
  concat = calloc(tailles + taillet + 1, sizeof(char));
  if (concat != NULL)
    {
      tmps = 0;
      tmpt = 0;
      tmpc = 0;
      while (s[tmps] != '\0')
        {
          concat[tmpc] = s[tmps];
          tmps++;
          tmpc++;
        }
      while (t[tmpt] != '\0')
        {
          concat[tmpc] = s[tmpt];
```

```
          tmpt++;
          tmpc++;
        }
      concat[tmpc] = '\0';
    }
  return concat;
}
```

13.5

```
int ordre(char *s, char *t)
{
  int tmps = 0, tmpt = 0;
  while ((s[tmps] == t[tmpt]) && (s[tmps] != '\0') && (t[tmpt] != '\0'))
    {
      tmps++;
      tmpt++;
    }
  if ((s[tmps] == '\0') && (t[tmpt] == '\0'))
    return 0;
  if ((s[tmps] != '\0') && (t[tmpt] == '\0'))
    return 1;
  if ((s[tmps] == '\0') && (t[tmpt] != '\0'))
    return -1;
  if (s[tmps] < t[tmpt])
    return -1;
  return 1;
}
```

13.6

```
#define TAILLE 1000
int espace()
{
  char *s;
  int tmp, esp = 0;
  s = calloc(TAILLE, sizeof(char));
  if (s == NULL)
    return -1;
  printf("Entrez une chaine :\n");
  fgets(s, TAILLE, stdin);
  tmp = 0;
  while (s[tmp] != '\0')
    {
      if (s[tmp] == ' ')
```

```
            esp++;
          tmp++;
        }
      return esp;
    }
```

13.7

```
#define TAILLE 1000
```

a)

```
int cherche(char *mot)
{
  FILE *fp;
  char buffer[TAILLE];
  int occ = 0;
  printf("Nom du fichier ? ");
  scanf("%s", buffer);
  fp = fopen(buffer, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture\n");
      return -1;
    }
  while (fscanf(fp, "%s", buffer) > 0)
    {
      if (strcmp(buffer, mot) == 0)
        occ++;
    }
  fclose(fp);
  return occ;
}
```

b)

```
int main()
{
  char m[TAILLE];
  printf("Entrez un mot :\n");
  scanf("%s", m);
  printf("%d occurrence(s) de %s dans le fichier\n", cherche(m), m);
  return 0;
}
```

13.8

```
#define TAILLE 1000
int somme(char *nom)
{
  FILE *fp;
  char buffer[TAILLE];
  int s = 0;
  fp = fopen(nom, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture\n");
      return -1;
    }
  while (fgets(buffer, TAILLE, fp) != NULL)
    {
      /* fgets lit aussi le retour à la ligne, il faut l'enlever */
      buffer[strlen(buffer) - 1] = '\0';
      s += strlen(buffer);
    }
  fclose(fp);
  return s;
}
```

13.9

```
#define TAILLE 1000
int debutLigne(char *nom)
{
  FILE *fp;
  char buffer[TAILLE];
  char texte[11] = "programmons";
  int nb = 0;
  fp = fopen(nom, "rt");
  if (fp == NULL)
    {
      printf("Erreur de lecture\n");
      return -1;
    }
  while (fgets(buffer, TAILLE, fp) != NULL)
    {
      if (strncmp(buffer, texte, sizeof(texte)) == 0)
        nb++;
    }
  fclose(fp);
  return nb;
}
```

FICHIERS BINAIRES 14

14.1 DIFFÉRENCE ENTRE FICHIERS TEXTE ET BINAIRE

Un fichier texte contient du texte *ASCII*. Lorsqu'un fichier texte contient des nombres, ces nombres sont codés sous forme de texte à l'aide des caractères `'1'`, `'2'`, etc. Dans ce format, chaque chiffre prend 1 octet en mémoire. On peut visualiser le contenu d'un fichier texte avec un éditeur de texte tel que *vi*, *emacs*, ou *notepad*. Les fonctions de lecture et écriture dans un fichier texte (`fscanf`, `fprintf`...) sont analogues aux fonctions de lecture et d'écriture de texte dans une console `scanf` et `printf`.

Un fichier binaire contient du code binaire. On ne peut pas visualiser son contenu avec un éditeur de texte. Lorsqu'une variable est écrite dans un fichier binaire, on écrit directement la valeur exacte de la variable, telle qu'elle est codée en binaire en mémoire. Cette manière de stocker les données est plus précise et plus compacte pour coder des nombres. Les fonctions de lecture et d'écriture dans un fichier binaire sont `fread` et `fwrite` qui lisent et écrivent des blocs de données sous forme binaire.

14.2 OUVERTURE ET FERMETURE D'UN FICHIER BINAIRE

De même que pour les fichiers texte, pour pouvoir utiliser les fichiers binaires, on doit inclure la bibliothèque d'entrées-sorties :

```
#include<stdio.h>
```

et déclarer pour chaque fichier un pointeur de fichier :

```
FILE *fp;  /* déclaration d'un pointeur de fichier fp */
```

On ouvre le fichier avec les modes `"r"`, `"w"`, `"a"`, `"r+"`, `"w+"` ou `"a+"`. Prenons le cas de l'ouverture d'un fichier en lecture seule :

```
FILE *fp;
fp = fopen("monfichier.dat","r");
```

Les différents modes possibles pour un fichier binaire sont :

- `"r"` : mode lecture seule. Le fichier est ouvert à son début prêt à lire les données. Toute tentative d'écriture dans le fichier provoque une erreur.
- `"w"` : mode écriture seule. Le fichier est initialement vide. Si le fichier existait avant, il est écrasé et les données sont perdues. Après l'appel de `fopen`, le fichier est prêt pour l'écriture de données. Toute tentative de lecture provoque une erreur.
- `"a"` : mode ajout. Le fichier n'est pas écrasé mais est prêt à écrire à la suite des données existantes. Toute tentative de lecture provoque une erreur.
- `"r+"` : mode lecture-écriture. Le fichier est prêt pour lire et écrire au début du fichier. Le fichier n'est pas écrasé. Lorsqu'on écrit une donnée, celle-ci remplace la donnée qui se trouvait éventuellement à cet emplacement sur le disque dans le fichier.
- `"w+"` : mode lecture-écriture. Le fichier est écrasé.
- `"a+"` : mode lecture-écriture. Le fichier n'est pas écrasé mais est prêt pour écrire à la suite des données existantes.

Après avoir utilisé un fichier, il faut le refermer en utilisant `fclose`. La fonction `fclose` prend en paramètre le pointeur de fichier et ferme le fichier.

14.3 LECTURE DANS UN FICHIER BINAIRE

Pour lire dans un fichier binaire, on lit en général dans le fichier les éléments d'un tableau. Chaque élément du tableau est appelé un *bloc*. Chaque bloc possède une **taille en octets**. Par exemple, un `char` correspond à 1 octet, un `float` correspond à 4 octets, etc.

La fonction `sizeof` donne la taille de chaque type. Par exemple, `sizeof(char)` vaut 1, et `sizeof(float)` vaut 4. On utilisera de préférence la fonction `sizeof` plutôt qu'une constante comme 1 ou 4 car cela augmente la lisibilité du programme et le programme ne dépend pas du compilateur ou du système. Par exemple la taille d'un `int` peut être soit 2 soit 4, mais `sizeof(int)` est toujours correct.

La fonction de lecture `fread` prend en paramètre le tableau, la taille de chaque bloc, le nombre de blocs à lire (nombre d'éléments du tableau), et le pointeur de fichiers. La taille physique du tableau doit être au moins égale au nombre de blocs lus, pour éviter une erreur mémoire. La fonction `fread` transfère les données du fichier binaire vers le tableau.

On peut **lire une variable** `x` avec la fonction `fread`. Il suffit pour cela de mettre l'adresse `&x` de la variable à la place du tableau et de mettre le nombre de blocs égal à 1 ; les données sont alors transférées dans la variable.



Compléments

√ D'une manière générale, l'adresse d'une variable peut toujours être considérée comme un tableau à un seul élément. Ceci est dû au fait qu'un tableau est seulement une adresse qui pointe vers une zone mémoire réservée pour le programme (statiquement ou dynamiquement).

La fonction `fread` retourne le nombre d'éléments effectivement lus. Si ce nombre est inférieur au nombre effectivement demandé, soit il s'est produit une erreur de lecture, soit la fin du fichier a été atteinte.

Exemple

Supposons qu'un fichier contienne le codage d'un tableau d'entiers que l'on va charger en mémoire centrale. Le fichier contiendra un `int` et des `float`. Le premier élément du fichier (de type `int`) donne le nombre d'éléments de type `float` qui se trouvent à la suite. Le programme suivant réalise le chargement d'un tableau en mémoire et son affichage dans la console.

```
#include <stdio.h>
#include <stdlib.h>

/* le nom du fichier est passé en paramètre (type char*) */
float* Chargement(char* nomFichier, int *adrNbElem)
{
   int n, ret;
   float *tableau;
   FILE *fp;
   if ((fp=fopen(nomFichier, "r")) == NULL)
      {
         printf("Erreur :");
         puts("fichier introuvable ou droits insuffisants");
         exit(1);
      }
   fread(&n sizeof(int), 1, fp); /* on lit le nombre d'éléments */
   *adrNbElem = n; /* passage par adresse */

   /* le nombre d'éléments connu, on alloue le tableau : */
   tableau = (float*)malloc(n*sizeof(float));  /* allocation */
   ret = fread(tableau, sizeof(float), n, fp);
   if (ret!=n)  /* lecture des éléments */
      puts("Erreur de lecture ou fin de fichier !");
   fclose(fp);  /* fermeture du fichier (obligatoire) */
   return tableau;
}
```

```
void Affichage(float* tableau, int nb)
{
   int i;
   for (i=0 ; i<nb ; i++)
      printf("%.3f ", tableau[i]);
}

int main(void)
{
   int nb;  /* nb n'est pas un pointeur mais un int */
   float *tableau;
   tableau = Chargement("monfichier.dat", &nb);
   Affichage(tableau, nb);
   free(tableau);  /* libération de mémoire */
   return 0;
}
```

14.4 Écriture dans un fichier binaire

Pour écrire dans un fichier binaire, on utilise la fonction `fwrite` qui transfère des données de la mémoire centrale vers un fichier binaire.

Comme la fonction `fread`, la fonction `fwrite` prend en paramètre le tableau, la taille de chaque bloc, le nombre de blocs à écrire et le pointeur de fichier. La taille physique du tableau doit être au moins égale au nombre de blocs écrits, pour éviter une erreur mémoire. La fonction `fwrite` transfère les données du tableau vers le fichier binaire.

La fonction `fwrite` retourne le nombre d'éléments effectivement écrits. Si ce nombre est inférieur au nombre effectivement demandé, il s'est produit une erreur d'écriture (fichier non ouvert, disque plein...).

Exemple

Le programme suivant lit au clavier un tableau de nombres réels et les sauvegarde dans un fichier binaire. Le format du fichier est le même que pour l'exemple précédent : on trouve d'abord le nombre d'éléments, puis les éléments à la suite dans le fichier.

```
#include <stdio.h>
#include <stdlib.h>

float* LectureTableau(int* adrNbElem)
{
   float *tableau;
   int i;
```

```
   puts("Entrez le nombre d'éléments");
   scanf("%d", adrNbElem);  /* passage par adresse, pas de  & */
   tableau = (float*)calloc(*adrNbElem, sizeof(float));
   for (i=0 ; i<*adrNbElem ; i++)
      scanf("%f", &tableau[i]);
   return tableau;
}

void Sauvegarde(int *tableau, int nb, char* nomFichier)
{
   FLIE *fp;
   if ((fp=fopen(nomFichier, "w")) == NULL)
      {
         puts("Permission refusée ou répertoire inexistant");
         exit(1);
      }
   /* écriture du nombre d'éléments */
   fwrite(&nb, sizeof(int), 1, fp)
   /* écriture des éléments */
   if (fwrite(tableau, sizeof(float), nb, fp)!=nb)
      puts("Erreur d'écriture dans le fichier !");
   fclose(fp);
}

int main(void)
{
   int nb;   /* ne pas mettre un pointeur ici */
   float* tableau;
   tableau = LectureTableau(&nb);
   Sauvegarde(tableau, nb, "monfichier.dat")
   free(tableau);   /* libération de mémoire */
   return 0;
}
```

14.5 Se positionner dans un fichier binaire

À chaque instant, un pointeur de fichier ouvert se trouve à une *position courante*, c'est-à-dire que le pointeur de fichier est prêt pour lire ou écrire à un certain emplacement dans le fichier. Chaque appel à `fread` ou `fwrite` fait avancer la position courante du nombre d'octets lus ou écrits.

La fonction `fseek` permet de se positionner dans un fichier, en modifiant la position courante pour pouvoir lire ou écrire à l'endroit souhaité. Lorsqu'on écrit sur un emplacement, la donnée qui existait éventuellement à cet emplacement est effacée et

remplacée par la donnée écrite. Le prototype de la fonction `fseek` permettant de se positionner est :

```
int fseek(FILE *fp, long offset, int origine);
```

La fonction modifie la position du pointeur fichier `fp` d'un nombre d'octets égal à `offset` à partir de l'`origine`. L'origine peut être :

- `SEEK_SET` : on se positionne par rapport au début du fichier ;
- `SEEK_END` : on se positionne par rapport à la fin du fichier ;
- `SEEK_CUR` : on se positionne par rapport à la position courante actuelle (position avant l'appel de `fseek`).

Exemple

L'exemple suivant traite de fichiers d'entiers. La fonction prend un entier *i* en paramètre permet à l'utilisateur de modifier le $(i+1)$ème entier du fichier.

```
void ModifieNombre(int i, FILE *fp)
{
   int n, nouveau;
   fseek(fp, i*sizeof(int), SEEK_SET);  /* positionnement */
   fread(&n, sizeof(int), 1, fp);           /* lecture */
   printf("L'ancien entier vaut %d\n", n);
   puts("Veuillez entrer la nouvelle valeur");
   scanf("%d", &nouveau);
   fseek(fp, -sizeof(int), SEEK_CUR);       /* recul d'une case */
   fwrite(&nouveau, sizeof(int), 1, fp);    /* écriture */
}
```

Exercices

14.1 (∗) Un fichier de nombres contient des `doubles`. Le fichier est structuré comme suit :

- La première donnée est un `int` qui donne le nombre de `doubles` du fichier.
- Les données suivantes sont les `doubles` les uns après les autres.

a) Écrire une fonction d'affichage des données du fichier. On passera le nom du fichier en paramètre.

b) Écrire une fonction de saisie des données du fichier. On ne fera pas intervenir de tableau. On passera le nom du fichier en paramètre.

c) Écrire une fonction qui affiche toutes les valeurs du fichier comprises entre deux nombres *a* et *b* passés en paramètre.

d) Écrire une fonction d'affichage de la *i*ème donnée du fichier.

e) Écrire une fonction qui affiche l'avant-dernière donnée du fichier si elle existe.



Compléments

√ Pour l'exercice suivant, on doit utiliser des pointeurs sur une structure. Si `p` est un pointeur sur une structures contenant un champ `x` de type `float`, alors `(*p)` est une structure, et `(*p).x` est un `float`. Pour alléger l'écriture, on peut utiliser la notation `p->x` à la place de `(*p).x` pour désigner le champ `x` de la structure pointée par `p`.

14.2 (∗∗) Un magasin d'articles de sport a une base de données pour gérer son fond. Chaque article est stocké en mémoire sous forme de structure :

```
typedef struct {
            int code;   /* code article */
            char denomination[100];  /* nom du produit */
            float prix; /* prix unitaire du produit */
            int stock;  /* stock disponible */
                 }TypeArticle;
```

La base de données est un fichier binaire dont :

- la première donnée est un entier qui représente le nombre d'articles de la base ;
- les données suivantes sont les structures représentant les différents articles de la base.

a) Écrire une fonction de chargement de la base de données en mémoire centrale sous forme de tableau de structures.

b) Écrire une fonction de sauvegarde de la base de données dans un fichier.

Dans toute la suite, on supposera que la base de données est chargée en mémoire sous forme de tableau.

c) Écrire une fonction de recherche du code d'un article à partir de sa dénomination.

d) Écrire une fonction d'affichage d'un article dont le code est passé en paramètre.

e) Écrire une fonction permettant à un utilisateur de modifier un article dont le code est passé en paramètre.

f) Écrire une fonction de saisie d'un nouvel article de la base de données.

g) Écrire une fonction de suppression d'un article de la base dont le code est passé en paramètre.

Corrigés

14.1

a)

```
void Affichage(char *nomFichier)
{
  int n, ret, i;
  double *tableau;
  FILE *fp;
  if ((fp = fopen(nomFichier, "rb")) == NULL)
    {
      puts("Erreur : fichier inexistant ou droits insuffisants");
      exit(1);
    }
  fread(&n, sizeof(int), 1, fp);
  printf("n=%d\n", n);
  tableau = (double *) malloc(n * sizeof(double));
  ret = fread(tableau, sizeof(double), n, fp);
  if (ret != n)
    {
      puts("erreur de lecture ou fin de fichier");
      exit(1);
    }
  fclose(fp);
  printf("Les éléments du tableau : \n");
  for (i = 0; i < n; i++)
    printf("%lg\n", tableau[i]);
  free(tableau);
}
```

b)

```
void Saisie(char *nomFichier)
{
  int n, i;
  double valeur;
```

```
  FILE *fp;
  if ((fp = fopen(nomFichier, "wb")) == NULL)
    {
      puts("Permission non accordée ou répertoire inexistant");
      exit(1);
    }
  printf("Entrez le nb d'éléments\n");
  scanf("%d", &n);
  fwrite(&n, sizeof(int), 1, fp);
  for (i = 0; i < n; i++)
    {
      printf("Entrer élément %d du tableau\n", i + 1);
      scanf("%lg", &valeur);
      fwrite(&valeur, sizeof(double), 1, fp);
    }
  fclose(fp);
}
```

c)

```
void Afficheab(char *nomFichier, double a, double b)
{
  double valeur;
  int n, ret, i;
  FILE *fp;
  if ((fp = fopen(nomFichier, "rb")) == NULL)
    {
      puts("Erreur : fichier inexistant ou droits insuffisants");
      exit(1);
    }
  ret = fread(&n, sizeof(int), 1, fp);
  if (ret < 1)
    {
      printf("Erreur de lecture ou fin du fichier\n");
      exit(1);
    }
  printf("Les éléments compris entre %lg et %lg :\n ", a, b);
  for (i = 0; i < n; i++)
    {
      ret = fread(&valeur, sizeof(double), 1, fp);
      if (valeur >= a && valeur <= b)
        printf("%lg\t", valeur);
    }
  puts("");
  fclose(fp);
}
```

d)

```
void Affichageieme(char *nomFichier, int i)
{
  double valeur;
  int n, ret;
  FILE *fp;
  if ((fp = fopen(nomFichier, "rb")) == NULL)
    {
      puts("Erreur : fichier inexistant ou droits insuffisants");
      exit(1);
    }
  fseek(fp, (1) * sizeof(int) + (i - 1) * sizeof(double), SEEK_SET);
  ret = fread(&valeur, sizeof(double), 1, fp);
  if (ret < 1)
    {
      printf("Erreur de lecture ou fin du fichier\n");
      fclose(fp);
      exit(1);
    }
  else
    printf("ième élément= %lg\n", valeur);
  fclose(fp);
}
```

e)

```
void AffichageAvantDernier(char *nomFichier)
{
  double valeur;
  int n, ret;
  FILE *fp;
  if ((fp = fopen(nomFichier, "rb")) == NULL)
    {
      puts("Erreur : fichier inexistant ou droits insuffisants");
      exit(1);
    }
  fseek(fp, -2 * sizeof(double), SEEK_END);
  ret = fread(&valeur, sizeof(double), 1, fp);
  if (ret < 1)
    {
      printf("Erreur de lecture ou fin du fichier\n");
    }
  else
    printf("Avant dernier élément= %lg\n", valeur);
}
```

14.2

```
typedef struct
{
  int code;
  char denomination[100];
  float prix;
  int stock;
} TypeArticle;
```

a)

```
TypeArticle *Chargement(char *nom_fichier, int *nb)
{
  FILE *fp;
  int i;
  TypeArticle *tab;
  fp = fopen(nom_fichier, "rb");
  if (fp == NULL)
    {
      fprintf(stderr, "Problème fopen");
      exit(1);

    }
  fread(nb, sizeof(int), 1, fp);
  printf("nb Chargement=%d\n", *nb);
  tab = (TypeArticle *) malloc((*nb) * sizeof(TypeArticle));
  for (i = 0; i < *nb; i++)
    {
      fread(&tab[i].code, sizeof(int), 1, fp);
      fread(tab[i].denomination, 100, 1, fp);
      fread(&tab[i].prix, sizeof(float), 1, fp);
      fread(&tab[i].stock, sizeof(int), 1, fp);
    } fclose(fp);
  for (i = 0; i < *nb; i++)
    {
      printf("%d\t%s\t%f\t%d\n", tab[i].code, tab[i].denomination,
             tab[i].prix, tab[i].stock);
    }
  return tab;
}
```

b)

```
void Sauvegarde(char *nomfichier)
{
  TypeArticle *tab;
  int i, nb;
```

```
  FILE *fp;
  if ((fp = fopen(nomfichier, "wb")) == NULL)
    {
      puts("permission non accordée ou répertoire inexistant");
      exit(1);
    }
  printf("Entrer le nombre d'articles\n");
  scanf("%d", &nb);
  tab = (TypeArticle *) malloc((nb) * sizeof(TypeArticle));
  fwrite(&nb, sizeof(int), 1, fp);
  for (i = 0; i < nb; i++)
    {
      printf("Entrer le code de l'article à rajouter\n");
      scanf("%d", &tab[i].code);
      printf("Entrer la nouvelle nomination\n");
      scanf("%s", &tab[i].denomination);
      fscanf(stdin, "%*c");
      tab[i].denomination[strlen(tab[i].denomination)] = '\0';
      printf("Entrer le nouveau prix \n");
      scanf("%f", &tab[i].prix);
      printf("Entrer le nouveau stock\n");
      scanf("%d", &tab[i].stock);
      fwrite(&tab[i].code, sizeof(int), 1, fp);
      fwrite(tab[i].denomination, 100, 1, fp);
      fwrite(&tab[i].prix, sizeof(float), 1, fp);
      fwrite(&tab[i].stock, sizeof(int), 1, fp);
    } free(tab);
  fclose(fp);
}
```

c)

```
int Recherche(TypeArticle * tableau, char *denomination, int nb)
{
  int i, cmp;
  for (i = 0; i < nb; i++)
    {
      cmp = strcmp(denomination, tableau[i].denomination);
      if (cmp == 0)
        return tableau[i].code;
    }
  return -1;
}
```

d)

```
void Affichearticlecode(TypeArticle * tableau, int code, int nb)
{
  int i, a;
  for (i = 0; i < nb; i++)
    {
      if (tableau[i].code == code)
        {
          printf("code %d : denomination=%s, prix=%f, stock=%d \n",
                 tableau[i].code, tableau[i].denomination,
                 tableau[i].prix, tableau[i].stock);
          break;
        }
    }
  printf("Le code recherché n'existe pas");
}
```

e)

```
int Modifiearticlecode(char *nomfichier, int code)
{
  int i, nb;
  TypeArticle Article;
  FILE *fp;
  if ((fp = fopen(nomfichier, "r+b")) == NULL)
    {
      puts("permission non accordée ou répertoire inexistant");
    }
  fread(&nb, sizeof(int), 1, fp);
  printf("nb=%d\n", nb);
  for (i = 0; i < nb; i++)
    {
      fread(&Article.code, sizeof(int), 1, fp);
      printf("code=%d\n", code);
      if (Article.code == code)
        {
          printf("Entrer la nouvelle nomination\n");
          scanf("%s", Article.denomination);
          fscanf(stdin, "%*c");
          Article.denomination[strlen(Article.denomination)] = '\0';
          printf("Entrer le nouveau prix\n");
          scanf("%f", &Article.prix);
          printf("Entrer le nouveau stock\n");
          scanf("%d", &Article.stock);
          fwrite(Article.denomination, 100, 1, fp);
          fwrite(&Article.prix, sizeof(float), 1, fp);
```

```
            fwrite(&Article.stock, sizeof(int), 1, fp);
            fclose(fp);
            return (0);
          }
        fseek(fp, (100 + sizeof(float) + sizeof(int)), SEEK_CUR);
      } printf("article non trouvé\n");
    fclose(fp);
    return 1;
}
```

f)

```
void Saisienouveauarticle(char *nomfichier)
{
    int i, a, nb;
    TypeArticle Article;
    FILE *fp;
    fp = fopen(nomfichier, "r+b");
    if (fp == NULL)
      {
        fprintf(stderr, "Problème fopen");
        exit(1);
      }
    fread(&nb, sizeof(int), 1, fp);
    printf("Saisienouveau article=%d\n", nb);
    printf("Entrer le code de l'article à rajouter\n");
    scanf("%d", &Article.code);
    printf("Entrer la nouvelle nomination\n");
    scanf("%s", Article.denomination);
    fscanf(stdin, "%*c");
    Article.denomination[strlen(Article.denomination) - 1] = '\0';
    printf("Entrer le nouveau prix\n");
    scanf("%f", &Article.prix);
    printf("Entrer le nouveau stock\n");
    scanf("%d", &Article.stock);
    nb = nb + 1;
    fseek(fp, 0, SEEK_SET);
    fwrite(&nb, sizeof(int), 1, fp);
    fseek(fp, 0, SEEK_END);
    fwrite(&Article.code, sizeof(int), 1, fp);
    fwrite(Article.denomination, 100, 1, fp);
    fwrite(&Article.prix, sizeof(float), 1, fp);
    fwrite(&Article.stock, sizeof(int), 1, fp);
    fclose(fp);
}
```

g)

```
int Suprimearticle(char *nomfichier, int code)
{
  int i = 0, nb, flag = 0;
  TypeArticle Article;
  FILE *fp;
  if ((fp = fopen(nomfichier, "r+b")) == NULL)
    {
      puts("permission non accordée ou répertoire inexistant");
    }
  fread(&nb, sizeof(int), 1, fp);
  while (i < nb && flag == 0)
    {
      fread(&Article.code, sizeof(int), 1, fp);
      if (Article.code == code)
        {
          flag = 1;
        }
      i++;
      fseek(fp, 100 + sizeof(float) + sizeof(int), SEEK_CUR);
  }
  if (flag == 0)
    {
      printf("article non trouvé\n");
      fclose(fp);
      return 1;
    }
  if (flag == 1 && i == nb) /* l'article à supprimer est le dernier */
    {
      fseek(fp, 0, SEEK_SET);
      nb = nb - 1;
      fwrite(&nb, sizeof(int), 1, fp);
      fclose(fp);
      return 1;
    }
  while (i < nb)
    {
      fread(&Article.code, sizeof(int), 1, fp);
      fread(Article.denomination, 100, 1, fp);
      fread(&Article.prix, sizeof(float), 1, fp);
      fread(&Article.stock, sizeof(int), 1, fp);
      fseek(fp, -2 * (sizeof(int) + 100 + sizeof(float) + sizeof(int)),
            SEEK_CUR);
      fwrite(&Article.code, sizeof(int), 1, fp);
```

```
      fwrite(Article.denomination, 100, 1, fp);
      fwrite(&Article.prix, sizeof(float), 1, fp);
      fwrite(&Article.stock, sizeof(int), 1, fp);
      i++;
      fseek(fp, 1 * (sizeof(int) + 100 + sizeof(float) + sizeof(int)),
            SEEK_CUR);
    } fseek(fp, 0, SEEK_SET);
  nb = nb - 1;
  fwrite(&nb, sizeof(int), 1, fp);
  fclose(fp);
  return 0;
}
```

Tableaux à double entrée 15

15.1 Tableaux de dimension 2

Un *tableau de dimension 2*, aussi appelé *tableau à double entrée*, ou *matrice*, est un tableau de tableaux. On peut déclarer un tableau de dimension 2 statiquement :

```
#define NB_LIGNES 100
#define NB_COLONNES 50
...
int tableau[NB_LIGNES][NB_COLONNES];
```

On accède aux différents éléments par des crochets : l'élément `tableau[i]` est un tableau, et son élément `j` est donc noté `tableau[i][j]`. Conventionnellement, on conçoit un tel tableau comme ayant des lignes et des colonnes, l'indice `i` représentant le numéro d'une ligne, et l'indice `j` représentant le numéro d'une colonne. L'élément `tableau[i][j]` se trouve à l'intersection de la ligne `i` et de la colonne `j`.

Exemple

Le programme suivant crée une matrice dont chaque élément (i, j) vaut $i + j$.

```
#define NB_LIGNES_MAX 100
#define NB_COLONNES_MAX 50
void Creer(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX],
           int nbl, int nbc)
{
   int i, j;
   for (i=0 ; i<nbl ; i++)   /* pour chaque ligne du tableau */
      for (j=0 ; j<nbc ; j++) /* pour chaque colonne */
         tableau[i][j] = i+j;
}
void Afficher(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX],
             int nbl, int nbc)
{
   int i, j;
   for (i=0 ; i<nbl ; i++)  /* pour chaque ligne */
      {
         for (j=0 ; j<nbc ; j++)  /* pour chaque colonne */
            printf("%d ", tableau[i][j]);
         printf("\n");  /* on va à la ligne */
      }
}
```

```
int main(void)
{
   int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX];
   int nbl, nbc;
   puts("Entrez le nombre de lignes et de colonnes");
   scanf("%d %d", &nbl, &nbc);
   if (nbl > NB_LIGNES_MAX || nbc > NB_COLONNES_MAX)
      {
         puts("Erreur, nombre dépassant la taille du tableau");
         exit(1);
      }
    Creer(tableau, nbl, nbc);
    Afficher(tableau, nbl, nbc);
    return 0;
}
```

Le résultat de l'exécution de ce programme pour 4 lignes et 6 colonnes est :

```
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
```

15.2 ALLOCATION DYNAMIQUE ET LIBÉRATION D'UN TABLEAU DE DIMENSION 2

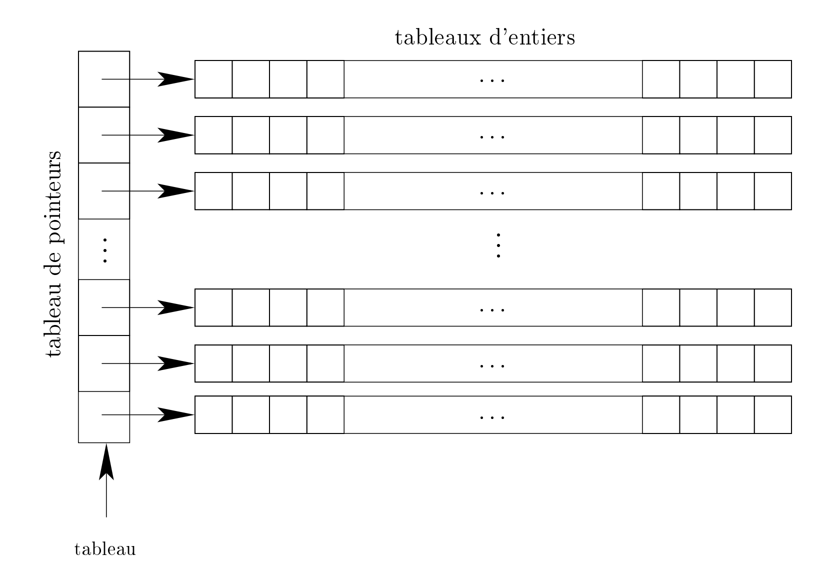
Un tableau (d'entiers) de dimension 2 avec allocation dynamique se déclare par un double pointeur :

```
int** tableau;
```

L'idée est qu'un tableau d'entiers est de type `int*` et un tableau de tableaux d'entiers est donc de type `(int*)*`.

Pour allouer le tableau de `nbl` lignes et `nbc` colonnes, on commence par allouer un tableau de `nbl` pointeurs. On alloue ensuite chacun de ces pointeurs avec `nbc` entiers dans une boucle (voir la figure 15.2).

```
/* La fonction retourne le tableau à double entrée */
int** Allocation(int nbl, int nbc)
{
   int **tab;
   int i;
   /* 1) allocation d'un tableau de nbl pointeurs */
   tab = (int**)calloc(nbl, sizeof(int*));
```




```
   /* 2) allocation d'un tableau de nbc entiers */
   /*    au bout de chaque pointeur */
   for (i=0 ; i<nbl ; i++)
      tab[i] = (int*)calloc(nbc, sizeof(int));
   return tab;
}
```

Pour libérer la mémoire d'un tableau de dimension 2 alloué dynamiquement, il faut libérer un à un tous les tableaux avec une boucle, puis libérer le tableau de pointeurs.

```
void Libération(int **tab, int nbl, int nbc)
{
   int i;
   for (i=0 ; i<nbl ; i++)
      if (nbc>0)
         free(tab[i]);
   if (nbl>0)
      free(tab);
}
```



Il faut toujours respecter le principe d'un appel à `free` pour chaque appel à `malloc` ou `calloc` pour libérer toute la mémoire.

Exercices

15.1 (∗)

a) Écrire une fonction de saisie d'un tableau à double entrée statique de `int`. On fera saisir le nombre de lignes et de colonnes au clavier.

b) Écrire une fonction qui prend en paramètre un tableau statique à double entrée de `int`, son nombre de lignes et son nombre de colonne, et compte le nombre de fois où apparaît un nombre x passé en paramètre.

c) Écrire une fonction qui prend en paramètre un tableau à double entrée `tab` et qui renvoie un tableau dont les éléments correspondent à la parité des éléments de `tab`.

d) Écrire un programme principal qui saisit un tableau à double entrée, et compte le nombre d'éléments qui sont pairs dans ce tableau.

15.2 (∗∗) On considère un tableau à m lignes et n colonnes entrées dans un fichier. La première ligne du fichier contient les nombres m et n. Les lignes suivantes du fichier contiennent les coefficients du tableau. Les colonnes sont séparées par des espaces.

```
m n
a_0,0    a_0,1    a_0,2     ...    a_0,n-1
a_1,0    a_1,1    a_1,2     ...    a_1,n-1
...      ...       ...      ...      ...
a_m-1,0  a_m-1,1  a_m-1,2   ...    a_m-1,n-1
```

a) Écrire une fonction d'allocation du tableau de dimension 2 de m lignes et n colonnes, les nombres m et n étant passés en paramètres.

b) Écrire une fonction de libération de mémoire pour un tableau de dimension 2.

c) Écrire une fonction qui réalise le chargement du fichier dans un tableau de tableaux (ou *matrice*) A.

d) Écrire une fonction qui calcule la somme des coefficients de chaque ligne de la matrice A et met tous les résultats dans un tableau de m nombres. La fonction retournera ce tableau.

e) Écrire le programme principal qui affiche la somme des coefficients de chaque ligne d'une matrice stockée dans un fichier, et libère la mémoire.

15.3 (∗∗) On représente un dictionnaire orthographique par un tableau de chaînes de caractères qui sont les mots qui sont dans le dictionnaire.

a) On suppose que les mots du dictionnaire sont stockés, un par ligne, dans un fichier. La première ligne du fichier contient le nombre de mots. Écrire une fonction de chargement du fichier en mémoire centrale sous forme de tableau de chaînes. On pourra suivre les étapes suivantes :

- lecture du nombre de mots et allocation d'un tableau de pointeurs.
- Pour chacun des mots
  - Lecture du mot ;
  - Allocation de la chaîne de caractères correspondante dans le tableau de chaînes ;
  - Recopie du mot dans la chaîne allouée.

b) On suppose que dans le fichier les mots sont classés dans l'ordre alphabétique. Écrire une fonction de recherche d'un mot dans le dictionnaire.

c) Écrire une fonction qui permet de rajouter un mot dans le dictionnaire. On pourra suivre les étapes suivantes :

- Saisie du nouveau mot au clavier ;
- Vérification que le mot n'est pas déjà dans le dictionnaire ;
- Pour agrandir la taille mémoire du dictionnaire :
  - Allocation d'un nouveau tableau de pointeurs ;
  - Faire pointer tous les pointeurs du nouveau tableau qui correspondent à des mots plus petits dans l'ordre alphabétique que le mot cherché sur la mémoire des mots de l'ancien dictionnaire ;
  - Allocation et copie du nouveau mot du dictionaire ;
  - Faire pointer tous les pointeurs du nouveau tableau qui correspondent à des mots plus grands dans l'ordre alphabétique que le mot cherché sur la mémoire des mots de l'ancien dictionnaire ;
  - Libérer l'ancien tableau de pointeurs et retourner le nouveau.

d) Écrire une fonction de sauvegarde d'un dictionnaire.

e) Écrire le programme principal qui charge un dictionaire, qui propose à l'utilisateur de rechercher ou d'ajouter des mots dans le dictionnaire ou de quitter, puis qui réalise la sauvegarde du dictionnaire en fin de programme.

Corrigés

15.1

```
#define NB_LIGNES_MAX 100
#define NB_COLONNES_MAX 50
```

a)

```
void Saisie(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX], int *nbl, int *nbc)
{
  int i, j;
  puts("Entrez le nombre de lignes et de colonnes");
  scanf("%d %d", nbl, nbc);
  for (i = 0; i < *nbl; i++)
    {
      printf("Entrez la ligne %d\n", i + 1);
      for (j = 0; j < *nbc; j++)
        scanf("%d", &tableau[i][j]);
    }
}
```

b)

```
void
CompteApparitionx(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX], int nbl,
                  int nbc, int x)
{
  int i, j, compteurx = 0;
  for (i = 0; i < nbl; i++)
    for (j = 0; j < nbc; j++)
      if (tableau[i][j] == x)
        compteurx++;
  printf("Le nombre %d apparaît %d fois\n", x, compteurx);
}
```

c)

```
int *tableauparite(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX],
                   int nbl, int nbc, int *nbtableau)
{
  int i, j;
  int *tabparite;
  *nbtableau = 0;
  tabparite = (int *) malloc(nbl * nbc * sizeof(int));
  for (i = 0; i < nbl; i++)
```

```
    for (j = 0; j < nbc; j++)
      if (tableau[i][j] % 2 == 0)
        {
          tabparite[*nbtableau] = tableau[i][j];
          (*nbtableau)++;
        }
  return tabparite;
}
```

d)

```
int
Compteparite(int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX], int nbl, int nbc)
{
  int i, j, nbtableau = 0;
  for (i = 0; i < nbl; i++)
    for (j = 0; j < nbc; j++)
      if (tableau[i][j] % 2 == 0)
        nbtableau++;
  return nbtableau;
}
```

programme principal

```
int main(void)
{
  int tableau[NB_LIGNES_MAX][NB_COLONNES_MAX];
  int nbl, nbc, nbtableau;
  if (nbl > NB_LIGNES_MAX || nbc > NB_COLONNES_MAX)
    {
      puts("Erreur, nombre dépassant la taille du tableau");
      exit(1);
    }
  Saisie(tableau, &nbl, &nbc);
  nbtableau = Compteparite(tableau, nbl, nbc);
  printf("le nombre d'éléments qui sont pairs dans le tableau= %d \n",
         nbtableau);
  return 0;
}
```

15.2

a)

```
int **Allocation(int m, int n)
{
  int **tab;
  int i;
```

```
  tab = (int **) calloc(m, sizeof(int *));
  for (i = 0; i < m; i++)
    tab[i] = (int *) calloc(n, sizeof(int));
  return tab;
}
```

b)

```
void Liberation(int **tab, int m, int n)
{
  int i;
  for (i = 0; i < m; i++)
    if (n > 0)
      free(tab[i]);
  if (m > 0)
    free(tab);
}
```

c)

```
int **Chargement(char *nomfichier, int *nbl, int *nbc)
{
  FILE *fp;
  int i, j;
  int **tab;
  fp = fopen(nomfichier, "rt");
  if (fp == NULL)
    {
      puts("Erreur d'ouverture du fichier");
      exit(1);
    }
  fscanf(fp, "%d %d", nbl, nbc);
  tab = Allocation(*nbl, *nbc);
  for (i = 0; i < *nbl; i++)
    {
      for (j = 0; j < *nbc; j++)
        fscanf(fp, "%d", &tab[i][j]);
    }
  fclose(fp);
  return tab;
}
```

d)

```
int *Sommecoefficient(int **tab, int nbl, int nbc)
{
  int i, j;
```

```
  int *tabsomme;
  tabsomme = (int *) calloc(nbl, sizeof(int));
  for (i = 0; i < nbl; i++)
    for (j = 0; j < nbc; j++)
      tabsomme[i] += tab[i][j];
  return tabsomme;
}
```

e)

programme principal

```
int main(void)
{
  int i, nbl, nbc;
  int **tab;
  int *sommecoeff;
  tab = Chargement("matrice.txt", &nbl, &nbc);
  sommecoeff = Sommecoefficient(tab, nbl, nbc);
  for (i = 0; i < nbl; i++)
    printf("somme des coeff. ligne %d = %d\n", i + 1, sommecoeff[i]);
  free(sommecoeff);
  Liberation(tab, nbl, nbc);
  return 0;
}
```

15.3

a)

```
char **Chargement(char *nomfichier, int *nbl)
{
  FILE *fp;
  int i, j;
  char **tab;
  char tmp[200];
  if ((fp = fopen(nomfichier, "rt")) == NULL)
    {
      puts("Erreur d'ouverture du fichier");
      exit(1);
    }
  fscanf(fp, "%d", nbl);
  tab = (char **) calloc(*nbl, sizeof(char *));
  for (i = 0; i < *nbl; i++)
    {
      fscanf(fp, "%s", tmp);
      tab[i] = (char *) calloc(strlen(tmp) + 1, sizeof(char));
```

```
            strcpy(tab[i], tmp);
        } fclose(fp);
    return tab;
}
```

b)

```
int Recherche(char **tab, int nbl, char *mot)
{
    int cmp, i;
    for (i = 0; i < nbl; i++)
        {
          cmp = strcmp(tab[i], mot);
          if (cmp == 0)
            return 0; /* mot trouvé */
          else if (cmp > 0)
            return 1; /* mot pas trouvé */
        }
    return 1;
}
```

c)

```
char **rajoutermot(char **tab, int *nbl)
{
    int j, i = 0, flag = 0;
    char mot[200];
    char **nouveautab;
    printf("Entrer le mot à ajouter\n");
    scanf("%s", mot);
    if (Recherche(tab, *nbl, mot) == 0)
        {
          printf("le mot existe déjà dans le dictionnaire");
          return NULL;
        }
    *nbl = *nbl + 1;
    nouveautab = (char **) calloc(*nbl, sizeof(char *));
    while (i < *nbl - 1 && flag == 0)
        {
          if (strcmp(tab[i], mot) < 0)
            {
              nouveautab[i] = tab[i];
              i++;
            }
          else
            flag = 1;
        }
```

```
  nouveautab[i] = (char *) calloc(strlen(mot) + 1, sizeof(char));
  strcpy(nouveautab[i], mot);
  i++;
  for (j = i; j < *nbl; j++)
    nouveautab[j] = tab[j - 1];
  if (*nbl > 0)
    free(tab);
  return nouveautab;
}
```

d)

```
void Sauvegarde(char *nomfichier, int nbl, char **nouveautab)
{
  FILE *fp;
  int i;
  char **tab;
  if ((fp = fopen(nomfichier, "wt")) == NULL)
    {
      puts("Erreur d'ouverture du fichier");
      exit(1);
    }
  fprintf(fp, "%d\n", nbl);
  for (i = 0; i < nbl; i++)
    fprintf(fp, "%s\n", nouveautab[i]);
  fclose(fp);
}
```

e)

```
 /*programme principal*/
int main(void)
{
  int i, motrecherche, nbl, choix;
  char **tab;
  char **nouveautab = NULL;
  char mot[200];
  tab = Chargement("dictionaire.txt", &nbl);
  while (choix != 0)
    {
      puts("\nMENU");
      puts("Pour rechercher un mot tapez 1");
      puts("Pour ajouter un mot tapez 2");
      puts("Pour quitter tapez 0");
      printf("\nVotre choix : ");
      scanf("%d", &choix);
```

```
      getchar();
      switch (choix)
        {
        case 1:
          puts("Rechercher un mot");
          printf("Entrer le mot à rechercher\n");
          scanf("%s", mot);
          motrecherche = Recherche(tab, nbl, mot);
          if (motrecherche == 0)
            printf("le mot est trouve\n");
          else
            printf("le mot n'est pas dans le dictionnaire\n");
          break;
        case 2:
          puts("Ajouter un mot\n");
          nouveautab = rajoutermot(tab, &nbl);
          break;
        case 0:
          puts("\nVous avez decidé de quitter... Au revoir !\n");
          break;
        }
    }
  if (nouveautab != NULL)
    {
      Sauvegarde("dictionaire.txt", nbl, nouveautab);
      for (i = 0; i < nbl; i++)
        free(nouveautab[i]);
      free(nouveautab);
    }
  return 0;
}
```

Partie 3

Algorithmes

LANGAGE ALGORITHMIQUE ET COMPLEXITÉ

16

16.1 POURQUOI UN AUTRE LANGAGE ?

L'informatique comprend de nombreux langages de programmation (*C*, *C++*, Java, Caml, Fortran, Pascal, Lisp, Prolog, Cobol, etc.). Programmer dans tous ces langage requiert une connaissance de base qui est indépendante du langage de programmation : l'algorithmique. Pour cela, on doit distinguer les *algorithmes*, qui sont des méthodes applicables par des ordinateurs pour résoudre différents problèmes, de leur *implémentation* dans tel ou tel langage particulier.

Nous étudions dans ce chapitre un langage algorithmique qui n'est pas un langage de programmation. Ce langage n'est pas très éloigné conceptuellement du langage *C* étudié jusqu'à maintenant, mais il permet tout de même de concevoir les algorithmes indépendamment de leur implémentation en *C*.

16.2 TYPES

Les types de base du langage sont :

- `entier` : c'est le type nombre entier, correspondant au type `int` du *C*, sauf qu'il n'est pas nécessairement codé sur 4 octets.
- `reel` : c'est le type nombre réel, pouvant correspondre avec les types `double` ou `float` du *C*.
- `caractere` : c'est le type caractère, correspondant au type `char` du *C*. Pour un tel `caractere` `c`, la fonction `ASCII(c)` donnera le code *ASCII* de `c`.

16.3 ENTRÉES-SORTIES

16.3.1 Clavier et écran

La fonction `lire` permet de lire une variable. Si `x` est une variable (de type entier, réel, caractère ou chaîne de caractères), l'instruction :

```
Lire(x);
```

permet de lire au clavier la valeur de `x`.

La fonction `ecrire` permet d'écrire le contenu d'une variable à l'écran. Si `x` est une variable (de type entier, réel, caractère ou chaîne de caractères), l'instruction :

```
ecrire(x);
```

permet d'afficher la valeur de `x`.

16.3.2 Fichiers texte

Le type `fichier` permet de lire et d'écrire dans un fichier. Pour cela, il faut d'abord déclarer et ouvrir un fichier :

```
Fichier f;
f ← ouvrir("nom.txt", "lect");
```

Les modes d'ouverture de fichiers peuvent être `"lect"` (lecture seule), `"ecr"` (écriture seule), ou bien le mode lecture-écriture `"lectEcr"`.

La fonction `lireF` permet de lire une variable dans un fichier. Si `x` est de type entier, réel, caractère ou chaîne de caractères, et f un fichier ouvert en lecture, l'instruction :

```
lireF(x, f);
```

met dans `x` la première valeur lue dans le fichier (et avance la position courante dans le fichier).

La fonction `ecrireF` permet d'écrire une variable dans un fichier texte. Si `x` est de type entier, réel, caractère ou chaîne de caractères, et `f` un fichier ouvert en écriture, l'instruction :

```
ecrireF(x, f);
```

permet d'écrire la valeur de `x` dans le fichier (et avance la position courante dans le fichier).

16.4 Syntaxe

L'affectation est notée ←, le test d'égalité est noté =.

Exemple 1

L'algorithme suivant calcule et affiche 2^n, où n est saisi au clavier.

```
Algorithme
début
   entier i, n, puiss;
   puiss ← 1;
   lire(n);
   pour i ← 0 à n-1 pas 1 faire
       puiss ← puiss * 2;
   fin faire
   ecrire(puiss);
fin
```

Exemple 2

Voici une version interactive de l'algorithme précédent :

```
Algorithme
début
   entier i, n, puiss;
   caractere choix;
   ecrire("Voulez-vous calculer une puissance de 2 ? ")
   lire(choix);
   si choix = 'y' faire
      puiss ← 1;
      i ← 0;
      lire(n);
      tant que i<n faire
          puiss ← puiss * 2;
          i ← i+1
      fin faire
      ecrire(puiss);
   fin faire
fin
```

16.5 Fonctions et procédures

16.5.1 Fonctions

Une fonction en langage algorithmique aura un prototype de la forme :

```
FONCTION NomFonction(Type1 nom1, type2 nom2,...): TypeRetour
```

où :

- `TypeRetour` est le type de la valeur retournée par la fonction ;
- `NomFonction` est le nom de la fonction ;
- `Type1`, `Type2`... sont les types respectifs des paramètres ;
- `nom1`, `nom2`... sont les identificateurs respectifs des paramètres.

Exemple 3

L'algorithme ci-dessus calculant 2^n peut être restructuré à l'aide d'une fonction :

```
FONCTION DeuxPuiss(entier n): entier
début
   entier i, puiss;
   puiss ← 1;
   pour i ← 0 à n-1 pas 1 faire
       puiss ← puiss * 2;
   fin faire
```

```
   retourner puiss;
fin

Algorithme
début
   entier n, puiss;
   lire(n);
   puiss ← DeuxPuiss(n);
   ecrire(puiss);
fin
```

16.5.2 Procédures

Une *procédure* est similaire à une fonction, mais elle ne retourne aucune valeur. Une procédure aura un prototype de la forme :

```
PROCEDURE NomFonction(Type1 nom1, typ2 nom2,...)
```

Exemple 4

Dans l'exemple suivant, la procédure `Affiche` permet l'affichage d'une donnée de type *entier*.

```
FONCTION DeuxPuiss(entier n): entier
début
   entier i, puiss;
   puiss ← 1;
   pour i ← 0 à n-1 pas 1 faire
       puiss ← puiss * 2;
   fin faire
   retourner puiss;
fin

PROCEDURE Affiche(entier a)
début
   ecrire("l'entier vaut : ");
   ecrire(a);
fin

Algorithme
début
   entier n, puiss;
   lire(n);
   puiss ← DeuxPuiss(n);
   Affiche(puiss);
fin
```

16.6 Enregistrements

Un ***enregistrement*** correspond à une structure *C*.

Exemple 5

```
enregistrement TypeArticle
   début
      entier code;
      caractere nom[100];
      reel prix;
   fin

PROCEDURE Affiche(TypeArticle A)
début
   ecrire(A.code);
   ecrire(A.nom);
   ecrire(A.prix);
fin

Algorithme
début
   TypeArticle A;
   ecrire("Entrez le code, le nom et le prix");
   lire(A.code, A.nom, A.prix);
   Affiche(A);
fin
```

16.7 Pointeurs, adresses et allocation

En langage algorithmique, les pointeurs et adresses fonctionnent comme en *C*.

```
entier n;
entier *p;
n ← 2;
p ← &n;
*p ← *p+1;
ecrire(n); /* affiche 3 */
```

La différence avec le *C* se situe au niveau des fonctions d'allocation. L'opérateur nouveau permet d'allouer de la mémoire en langage algorithmique.

Exemple 6

```
enregistrement TypeArticle
   début
      entier code;
```

```
      caractere nom[100];
      reel prix;
   fin

FONCTION AlloueTableau(entier* adr_n): TypeArticle*
début
   TypeArticle *resultat;
   entier n;
   ecrire("Entrez le nombre d'éléments : ");
   lire (n);
   *adr_n = n;
   /* allocation d'un tableau de n TypeArticle */
   resultat ← nouveau TypeArticle[n];
   retourner resultat;
fin
```

16.8 Notion de complexité d'un algorithme

16.8.1 Définition intuitive de la complexité

La *complexité* d'un algorithme est une estimation du nombre d'opérations de base effectuées par l'algorithme en fonction de la taille des données en entrée de l'algorithme.

Par exemple, si un algorithme écrit dans une fonction prend en entrée un tableau de n éléments, la complexité de l'algorithme sera une estimation du nombre total d'opérations de base nécessaires pour l'algorithme, en fonction de n. Plus n sera grand, plus il faudra d'opérations. La nature de l'algorithme sera différente selon que sa complexité sera plutôt de l'ordre de n, de n^2, de n^3, ou bien de 2^n. Le temps de calcul pris par l'algorithme ne sera pas le même.

Une opération de base pourra être une affectation, un test, une incrémentation, une addition, une multiplication, etc.

16.8.2 Notion de grand O

On appelle *opération de base*, ou *opération élémentaire*, toute affectation, test de comparaison $=, >, <, \leq, \ldots$, opération arithmétique $+, -, *, /$, appel de fonctions comme `sqrt`, incrémentation, décrémentation. Lorsqu'une fonction ou une procédure est appelée, le coût de cette fonction ou procédure est le **nombre total d'opéra-**

tions élémentaires engendrées par l'appel de cette fonction ou procédure. Le temps de calcul pris par l'algorithme (sur une machine donnée) est directement lié à ce nombre d'opérations.

En fait, il est hors de question de calculer exactement le nombre d'opérations engendrées par l'application d'un algorithme. On cherche seulement un **ordre de grandeur** (voir figure 16.1). L'ordre de grandeur asymptotique est l'ordre de grandeur lorsque la taille des données devient très grande.

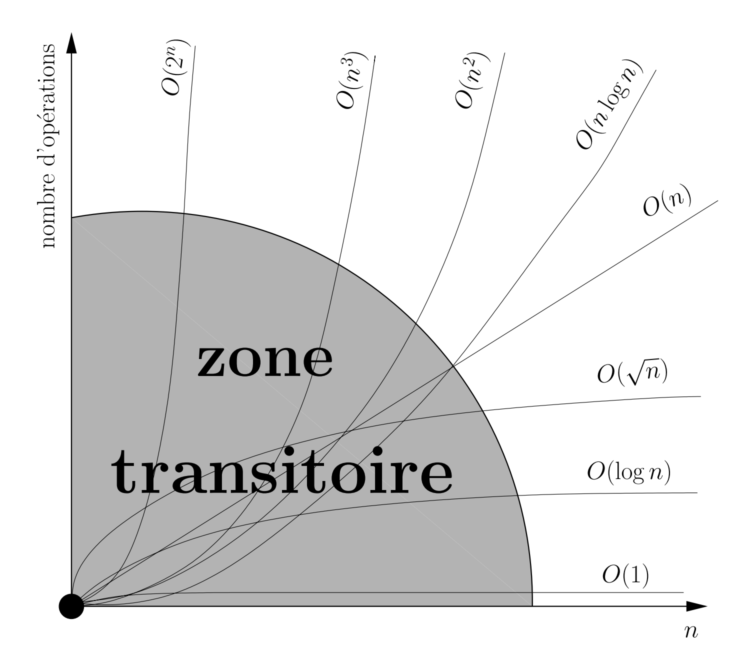


Figure 16.1– **Ordres de grandeurs asymptotiques**

Soit un algorithme dépendant d'une donnée d de taille n (par exemple un tableau de n éléments). Notons $NB = (d, n)$ le nombre d'opérations engendrées par l'algorithme.

On dit que l'algorithme est en $O(n)$ si et seulement si on peut trouver un nombre K tel que (pour n assez grand) :

$$NB(d, n) \leq K.n$$

Dans ce cas, on dit aussi que l'algorithme est *linéaire*.

On dit que l'algorithme est en $O(n^2)$ si et seulement si on peut trouver un nombre K tel que (pour n assez grand) :

$$NB(d, n) \leq K.n^2$$

Dans ce cas, on dit aussi que l'algorithme est *quadratique*.

On dit que l'algorithme est en $O(2^n)$ si et seulement si on peut trouver un nombre K tel que (pour n assez grand)

$$NB(d, n) \leq K.2^n$$

Dans ce cas, on dit aussi que l'algorithme est *exponentiel*.

Exercices

Pour chacun des exercices de cette partie, **on évaluera la complexité** de l'algorithme proposé.

16.1 (∗) Écrire un algorithme qui initialise un tableau de n entiers, le nombre n étant saisi au clavier, la valeur d'indice i du tableau valant 1 si $3i^2 + i - 2$ est multiple de 5 et 0 sinon.

16.2 (∗) Écrire une fonction en langage algorithmique qui prend en paramètre un entier i et calcule la valeur $u(i)$ définie par récurrence par :

$$u_0 = 1\ ;\ u_1 = 2$$

$$u_{j+1} = 3 * u_j - u_{j-1} \text{ pour } j \geq 1$$

16.3 (∗) Écrire une fonction en langage algorithmique qui saisit les éléments d'un tableau dont le nombre d'éléments n est passé en paramètre. On supposera que le tableau a déjà été alloué et est passé en paramètre.

16.4 (∗) Écrire une fonction en langage algorithmique qui prend en paramètre un entier n et un tableau à deux dimensions de taille $n \times n$ d'entiers. La fonction calcule la somme des éléments du tableau.

16.5 (∗∗) Écrire une fonction en langage algorithmique qui calcule les valeurs d'un tableau T de taille n, le nombre n étant saisi au clavier, dont les valeurs $T[i]$ sont telles que $T[0] = 1$ et pour $i \geq 1$:

$$T[i] = i * (T[0] + T[1] + \cdots + T[i-1])$$

16.6 (∗∗) Soit la fonction

$$f(x) = \begin{cases} 3x^2 + x + 1 & \text{si } x \geq 1 \\ 0 & \text{sinon} \end{cases}$$

a) Écrire une fonction en langage algorithmique qui prend en paramètre un entier x et calcule $f(x)$.

b) Écrire une fonction en langage algorithmique qui prend en paramètre un entier n et calcule la somme :

$$S_n = f(n) + f(n-1) + f(n-2) + f(n-3) + f(n-4) + \cdots$$

On arrêtera la somme lorsque la valeur f calculée est nulle.

c) Écrire une fonction en langage algorithmique qui prend en paramètre un entier n et calcule la somme :

$$T_n = f(n) + f(n/2^2) + f(n/3^2) + f(n/4^2) + f(n/5^2) + \cdots$$

Dans cette somme, les divisions sont des divisions euclidiennes. On arrêtera la somme lorsque la valeur f calculée est nulle.

d) Écrire une fonction en langage algorithmique qui prend en paramètre un entier n et calcule la somme :

$$U_n = f(n) + f(n/2) + f(n/4) + f(n/8) + f(n/16) + \cdots$$

Dans cette somme, les divisions sont des divisions euclidiennes. On arrêtera la somme lorsque la valeur f calculée est nulle.

16.7 (∗ ∗ ∗) (Recherche dichotomique) On cherche à savoir si un tableau de n nombres entiers contient une certain nombre a. On suppose que le tableau est trié dans l'ordre croissant, c'est-à-dire que chaque élément est inférieur ou égal au suivant dans le tableau.

On pourrait évidemment tester tous les nombres du tableau en les comparant à a, mais cela nécessiterait (dans le pire des cas) n comparaisons. On se propose d'appliquer une méthode qui nécessite moins d'opération. La méthode s'appelle la recherche dichotomique.

À chaque étape, on recherche un élément égal à a entre deux indices i_{min} et i_{max}. Au départ, l'élément peut se trouver n'importe où (entre $i_{min} = 0$ et $i_{max} = n - 1$).

On compare l'élément d'indice $i = (i_{min} + i_{max})/2$ avec a. Trois cas peuvent se produire :

- Si l'élément d'indice i est égal à a, l'algorithme se termine : le nombre a se trouve bien dans le tableau.

- Si l'élément d'indice i est supérieur à a, un élément égal à a ne peut se trouver qu'à un indice plus petit que i, puisque le tableau est trié. Le nouvel i_{max} vaut $i - 1$.
- Si l'élément d'indice i est inférieur à a, un élément égal à a ne peut se trouver qu'à un indice plus grand que i. Le nouvel i_{min} vaut $i + 1$.

	éléments $\leq$ tab[i]		éléments $\geq$ tab[i]	
i_{min}		i		i_{max}

Á chaque étape, un élément égal à a ne peut se trouver qu'entre i_{min} et i_{max}. On itère ce procédé jusqu'à ce que i_{min} devienne strictement supérieur à i_{max}. Il ne reste plus qu'à tester un seul élément.

Comme à chaque étape le nombre d'éléments est divisé par 2, le nombre k d'étapes est tel que $\frac{n}{2^k} \geq 1$. Donc $k \leq \log_2(n)$.

a) Écrire une de recherche dichotomique d'un nombre x dans un tableau de n nombres. La fonction renvoie 1 si l'élément se trouve dans le tableau et 0 sinon.

b) Quelle est la complexité ?

Corrigés

16.1

```
Algorithme
début
   entier i, n, puiss;
   entier *tab;
   lire(n);
   tab ← nouveau entier[n];
   pour i ← 0 à n-1 pas 1 faire
       si (3*i*i+i-2)%5 = 0  faire
       tab[i] ← 1;
       sinon faire
       tab[i] ← 0;
   fin faire
fin
```

La complexité est de $O(n)$

16.2

```
FONCTION Calculu(entier i): entier
début
   entier j;
   entier U0, U1, U2;
   U0 ← 1;
   U1 ← 2;
   si i=0  faire
       retourner U0;
   fin faire
   si i=1  faire
       retourner U1;
   fin faire
   j ← 2;
   tant que j<=i faire
          U2 ← 3 * U1-U0;
  U0 ← U1;
          U1 ← U2;
          j ← j+1;
   fin faire
   retourner U2;
fin
```

La complexité est de $O(i)$

16.3

```
PROCEDURE Saisit(entier n, entier *tab)
début
   entier i;
   pour i ← 0 à n-1 pas 1 faire
   lire (tab[i]);
   fin faire
fin
```

La complexité est de $O(n)$

16.4

```
FONCTION Somme(entier n,entier **tab): entier
début
   entier i, j;
   entier somme ← 0;
   pour i ← 0 à n-1 pas 1 faire
    pour j ← 0 à n-1 pas 1 faire
```

```
     somme ← somme + tab[i][j];
     fin faire
    fin faire
    retourner somme;
fin
```

La complexité est de $O(n^2)$

16.5

```
FONCTION CalculerT(entier *nb): entier *
début
    entier n, i;
    entier *tab;
    lire(n);
    *nb ← n;
    tab ← nouveau entier[n];
    tab[0] ← 1;
    pour i ← 0 à n-1 pas 1 faire
     tab[i] ← 0;
     pour j ← 0 à i-1 pas 1 faire
     tab[i] ← tab[i] + tab[j];
     fin faire
     tab[i] ← tab[i]*i;
    fin faire
    retourner tab;
fin
```

La complexité est de $O(n^2)$

16.6

a)

```
FONCTION CalculerFx(entier x): entier
début
si x<1  faire
        retourner 0;
  fin faire
  retourner 3*x*x+x+1;
fin
```

La complexité est de $O(1)$

b)

```
FONCTION CalculerSn(entier n): entier
début
entier j ← n;
```

```
  entier S ← 0;
  tant que j>0 faire
          S ← S + CalculerFx(j);
          j ← j-1;
  fin faire
  retourner S;
fin
```

La complexité est de $O(n)$

c)

```
FONCTION CalculerTn(entier n): entier
début
entier i, j, Fx;
entier T ← 0;
  i ← n;
  j ← 2;
Fx ← CalculerFx(i);
  tant que Fx !=0 faire
          T ← T + Fx;
          i ← n/(j*j);
          j ← j+1;
  Fx ← CalculerFx(i);
  fin faire
  retourner T;
fin
```

La complexité est de $O(\sqrt{n})$

d)

```
FONCTION CalculerUn(entier n): entier
début
entier i, j, Fx;
entier U ← 0;
  i ← n;
  j ← 1;
Fx ← CalculerFx(i);
  tant que Fx !=0 faire
          U ← U + Fx;
          j ← j * 2;
          i ← n/j;
  Fx ← CalculerFx(i);
  fin faire
  retourner U;
fin
```

La complexité est de $O(log_2(n))$

16.7

a)

```
FONCTION Recherchedicho(entier a, entier *tab, entier n): entier
début
   entier debut ← 0, fin ← n-1, milieu;
   tant que debut <=fin faire
milieu ← (debut+fin)/2;
si a= tab[milieu]  faire
retourner 1;
si a < tab[milieu]  faire
fin=milieu-1;
sinon faire
debut=milieu + 1;
  fin faire
  retourner 0;
fin
```

b) La complexité est de $O(log_2(n))$

ALGORITHMES DE TRI QUADRATIQUES

17

17.1 QU'EST-CE QU'UN TRI ?

Donnons-nous un tableau de nombres. Le problème est de trier ces nombres dans l'ordre croissant. En d'autres termes, nous devons modifier le tableau pour qu'il contienne les même nombres qu'auparavant, mais cette fois les nombres sont rangés dans l'ordre croissant, du plus petit au plus grand. Ce problème est un grand classique d'algorithmique.

Exemple

Le résultat du tri du tableau :

6	3	7	2	3	5

est le suivant :

2	3	3	5	6	7

Plus généralement, on peut chercher à trier des objets sur lesquels est définie une relation d'ordre $\leq$. On peut ainsi trier des objets suivant un critère (trier des gens suivant leur taille, trier des mots dans l'ordre alphabétiques, etc.).

17.2 TRI PAR SÉLECTION

17.2.1 Principe du tri par sélection

Donnons-nous un tableau `T` de `n` nombres. Le principe du tri par sélection est le suivant. On sélectionne le maximum (le plus grand) de tous les éléments, et on le place en dernière position `n-1` par un échange. Il ne reste plus qu'à trier les `n-1` premiers éléments, pour lesquels on itère le procédé.

Exemple

Soit le tableau

6	3	7	2	3	5

Le maximum de ce tableau est 7. Plaçons le 7 en dernière position par un échange.

6	3	5	2	3	7

reste à trier

Le 7 est à sa position définitive ; il ne reste plus qu'à trier les cinq premiers éléments. Le maximum de ces cinq éléments est 6. Plaçons-le à la fin par un échange :

3	3	5	2	6	7

reste à trier

Le maximum parmi les nombres restant à trier est 5. Plaçons-le en dernière position par un échange :

3	3	2	5	6	7

à trier

Enfin le maximum parmi les nombres à trier est 3. On le place en dernier et le tableau est trié.

2	3	3	5	6	7

17.2.2 Algorithme de tri par sélection

L'algorithme de tri par sélection (par exemple sur des nombres entiers) est le suivant :

```
PROCEDURE TriSelection(entier *T, entier n)
début
   entier k, i, imax, temp;
   pour k ← n-1 à 1 pas -1 faire
      /* recherche de l'indice du maximum : */
      imax ← 0;
      pour i ← 1 à k pas 1 faire
         si T[imax] < T[i]  faire
            imax ← i;
         fin faire
      fin faire
      /* échange : */
      temp ← T[k];
      T[k] ← T[imax];
      T[imax] ← temp;
   fin faire
fin
```

17.2.3 Estimation du nombre d'opérations

- **Coût des échanges.** Le tri par sélection sur n nombres fait $n - 1$ échanges, ce qui fait $3(n - 1)$ affectations.

- **Coût des recherches de maximum :**
  - On recherche le maximum parmi n éléments : au plus $4(n-1)$ opérations. (c'est le nombre d'itération de la boucle sur `i` pour `k` fixé égal à `n-1`)
  - On recherche ensuite le maximum parmi $n-1$ éléments : au plus $4(n-2)$ tests. (c'est le nombre d'itération de la boucle sur `i` pour `k` fixé égal à `n-2`)
  - On recherche ensuite le maximum parmi $n-2$ éléments : $4(n-3)$ tests.
  - ...

  Le nombre total de tests est de :

$$4\,(1+2+3+\cdots+(n-2)+(n-1)) = 4\frac{(n-1)n}{2}$$

C'est la somme des termes d'une suite arithmétique.
Au total, le nombre d'opérations est plus petit que $3(n-1)+4\frac{(n-1)n}{2}$, qui est un polynôme de degré 2 en le nombre n d'éléments du tableau. On dit que la méthode est *quadratique*, ou encore qu'elle est une méthode en $O(n^2)$. La partie de degré 1 en n est négligeable devant la partie en n^2 quand n devient grand.

17.3 Tri par insertion

17.3.1 Principe du tri par insertion

Le principe du tri par insertion est le suivant : on trie d'abord les deux premiers éléments, puis on insère le 3ème à sa place pour faire une liste triée de 3 éléments, puis on insère le 4ème élément dans la liste triée. La liste triée grossit jusqu'à contenir les n éléments.

Exemple

Soit le tableau

6	3	4	2	3	5

En triant les deux premiers éléments on obtient :

3	6	4	2	3	5

trié

En insérant le troisième élément à sa place dans la liste triée :

3	4	6	2	3	5

trié

En insérant le quatrième élément à sa place dans la liste triée :

2	3	4	6	3	5

trié

En insérant le cinquième élément à sa place dans la liste triée :

2	3	3	4	6	5

trié

En insérant le sixième élément à sa place dans la liste triée :

2	3	3	4	5	6

Pour insérer un élément à sa place, on doit décaler les éléments déjà triés qui sont plus grands.

17.3.2 Algorithme de tri par insertion

L'algorithme de tri par insertion est le suivant :

```
PROCEDURE TriInsertion(entier *T, entier n)
début
   entier k, i, v;
   pour k ← 1 à n-1 pas 1 faire
      v ← T[k];
      i ← k-1;
      /* On décale les éléments pour l'insertion */
      tant que i ≥ 0 et v < T[i] faire
            T[i+1] ← T[i];
            i ← i-1;
       fin faire
       /* On fait l'insertion proprement dite */
       T[i+1] ← v;
   fin faire
fin
```

17.3.3 Estimation du nombre d'opérations

L'indice k varie de 1 à $n-1$, soit $n-1$ valeurs différentes. Pour chaque valeur de k, l'indice i prend **au plus** k valeurs différents. Le total du nombre d'opérations est donc au plus de :

$$3(n-1) + \underbrace{4\,(1+2+3+\cdots+(n-2)+(n-1))}_{\text{somme pour différentes valeurs de } k}$$

Soit un nombre d'opération d'au plus :

$$3(n-1) + 4\frac{(n-1)n}{2}$$

ce qui est un polynôme du second degré en le nombre n d'éléments. L'algorithme est donc quadratique ($O(n^2)$).

17.4 Tri par bulles

17.4.1 Principe du tri par bulles

Dans le tri par bulle, on échange deux éléments successifs `T[i-1]` et `T[i]` s'ils ne sont pas dans l'ordre. Évidemment, il faut faire cela un grand nombre de fois pour obtenir un tableau trié. Plus précisément, on fait remonter le maximum à sa place définitive par échanges successifs avec ses voisins.

Exemple

Voici la succession des échanges d'éléments successifs effectués sur un exemple.

6	3	4	2	3	5

3	6	4	2	3	5

3	4	6	2	3	5

3	4	2	6	3	5

3	4	2	3	6	5

3	4	2	3	5	6

On fait une deuxième passe :

3	2	4	3	5	6

3	2	3	4	5	6

Puis encore une passe :

2	3	3	4	5	6

17.4.2 Algorithme du tri par bulles

```
PROCEDURE TriBulle(entier *T, entier n)
début
   entier i, k, temp;
   /* pour chaque passe */
   pour k ← n-1 à 1 pas -1 faire
```

```
      /* on fait remonter le plus grand */
      pour i ← 1 à k pas 1 faire
         si T[i] < T[i-1]  faire
            /* échange de T[i-1] et T[i] */
            temp ← T[i];
            T[i] ← T[i-1];
            T[i-1] ← temp;
         fin faire
      fin faire
   fin faire
fin
```

17.4.3 Estimation du nombre d'opérations

La variable k prend $n - 1$ valeurs différentes. Pour chacune de ces valeurs la variable i prend k valeurs différentes. Le nombre total d'opérations est plus petit que :

$$2(n-1) + 5\underbrace{(1 + 2 + 3 + \cdots + (n-2) + (n-1))}_{\text{somme pour différentes valeurs } k}$$

Ce qui fait un nombre d'opération au plus :

$$2(n-1) + 5\frac{(n-1)n}{2}$$

qui est un polynôme de degré 2 en le nombre n d'éléments du tableau. L'algorithme est donc quadratique ($O(n^2)$).

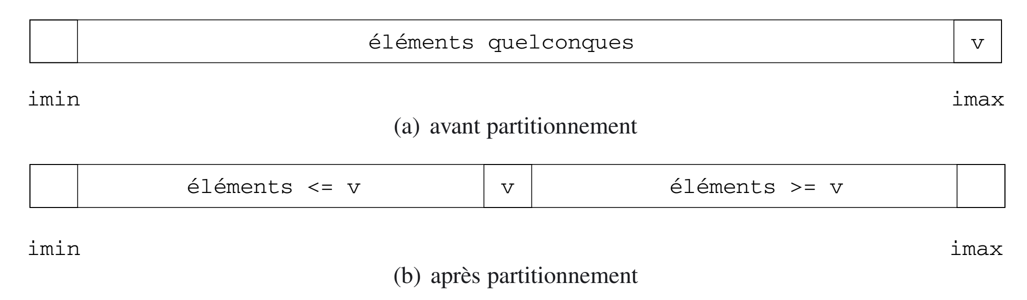
Le tri rapide (*quicksort*)

18

Le tri rapide fonctionne suivant la méthode *diviser pour régner*. On partage le tableau en deux, la partie gauche et la partie droite, et on trie chacune des deux parties séparément. On obtient alors un tableau trié.

18.1 Partitionnement

Supposons que l'on souhaite trier **une partie d'un tableau** `T` entre les indices `imin` et `imax`. On choisit une valeur pivot, par exemple `v = T[imax]`. On veut séparer dans le tableau les éléments inférieurs à `v` et les éléments supérieurs à `v`.



(a) avant partitionnement

(b) après partitionnement

Figure 18.1 – **Partitionnement dans le tri rapide**

Pour faire cela, on introduit un indice `i`, initialisé à `imin`, et un indice `j`, initialisé à `imax-1`. On fait ensuite remonter l'indice `i` jusqu'au premier élément supérieur à `v`, et on fait redescendre `j` jusqu'au premier élément inférieur à `v`. On échange alors les éléments d'indices `i` et `j`. On itère ce procédé tant que $i \leq j$. À la fin, on place la valeur pivot en position `i`.

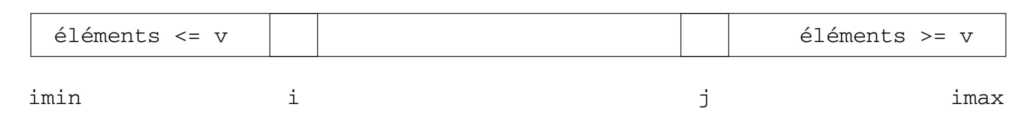


Figure 18.2 – **Échange des éléments d'indice `i` et `j` s'il y a lieu**

Le code algorithmique de ce partitionnement est donc :

```
FONCTION Partitionnement(entier *T, entier imin, entier imax)
                            : entier
début
   entier v, i, j, temp;
```

```
    v ← T[imax];        /* v valeur pivot */
    i ← imin;
    j ← imax-1;
    tant que i<=j faire
       tant que i<imax et T[i] <= v faire
          i ← i+1;
       fin faire
       tant que j>=imin et T[j] >= v faire
          j ← j-1;
       fin faire
       si i<j  faire   /* échange */
          temp ← T[i];
          T[i] ← T[j];
          T[j] ← temp;
       fin faire
    fin faire
    T[imax] ← T[i]; T[i] ← v;  /* On place la valeur pivot en i */
    retourner i;  /* renvoie l'endroit de la séparation */
fin
```

18.2 L'ALGORITHME DE TRI RAPIDE

Après le partitionnement d'un tableau, le pivot v est à sa place définitive dans le tableau trié. De plus, tous les éléments qui se trouvent à la gauche du pivot dans le tableau triés sont ceux qui se trouvent à sa gauche après partitionnement. Le tri rapide consiste à effectuer le partitionnement du tableau en deux parties, puis à trier chacune des deux parties (gauche et droite) suivant le même principe.

```
PROCEDURE TriRapide(entier *T, entier imin, entier imax)
début
   entier i;
   si imin < imax  faire
      i ← Partitionnement(T, imin, imax);
      TriRapide(T, imin, i-1);  /* tri de la partie gauche */
      TriRapide(T, i+1, imax);  /* tri de la partie droite */
   fin faire
fin

PROCEDURE QuickSort(entier *T, entier n)
début
   TriRapide(T,0, n-1);  /* imin=0  et  imax=n-1 */
fin
```

Notons que la procédure `TriRapide` s'appelle elle-même. On parle de procédure *récursive*. Nous étudierons en détail cette manière de programmer dans un autre chapitre.

18.3 Comparaison de temps de calcul

Voici (figure 18.3) des courbes expérimentales qui montrent le temps de calcul (temps d'utilisation du *CPU*) des algorithmes de tri par bulle, par insertion, par sélection, et du tri rapide, en fonction de la taille n du tableau. Le nombre d'éléments n du tableau varie de 1 à 2000. On observe que le temps de calcul donné par le tri rapide est nettement meilleur que le temps de calcul des tris quadratiques (la courbe est collée près de l'axe des x). La complexité se traduit au niveau des temps de calcul.

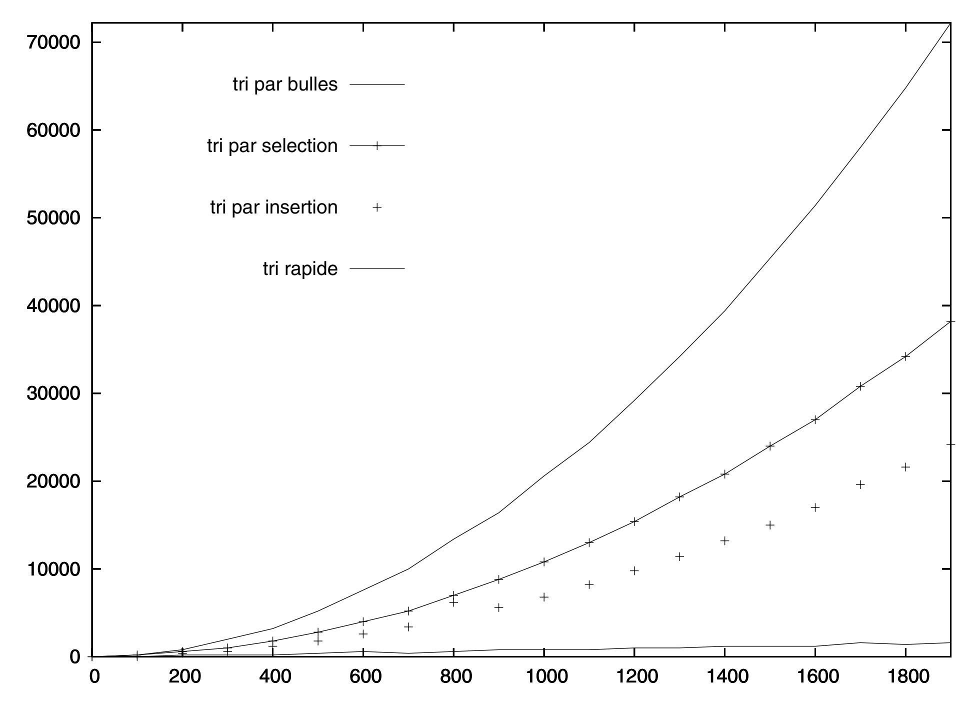


Figure 18.3 – Comparaison expérimentale de temps de calcul d'algorithmes de tri

Exercices

18.1 (∗) Appliquer la méthode de partitionnement sur le tableau suivant :

7	9	2	3	6	7	4	3	8	1	5

18.2 (∗) Appliquer la méthode de partitionnement sur le tableau suivant :

3	1	7	8	6	5	4	3	6	7	6

18.3 (∗) Appliquer le tri rapide pour trier le tableau suivant :

7	2	6	9	3	5	3	1	4

18.4 (∗) Appliquer le tri rapide pour trier le tableau suivant :

7	9	2	3	6	7	4	3	8	1	5

18.5 (∗) Appliquer le tri rapide pour trier le tableau suivant :



3	1	7	8	6	5	4	3	6	7	6

Corrigés

18.1

```
1 3 2 3 4 5 6 9 8 7 7
```

18.2

```
3 1 3 4 6 5 6 7 6 7 8
```

18.3

```
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
1 2 3 3 4 5 6 7 9
```

18.4

```
1 3 2 3 4 5 6 9 8 7 7
1 3 2 3 4 5 6 9 8 7 7
1 3 2 3 4 5 6 9 8 7 7
1 2 3 3 4 5 6 9 8 7 7
1 2 3 3 4 5 6 7 8 7 9
1 2 3 3 4 5 6 7 8 7 9
1 2 3 3 4 5 6 7 7 8 9
1 2 3 3 4 5 6 7 7 8 9
```

18.5

```
3 1 3 4 6 5 6 7 6 7 8
3 1 3 4 5 6 6 7 6 7 8
3 1 3 4 5 6 6 7 6 7 8
3 1 3 4 5 6 6 7 6 7 8
1 3 3 4 5 6 6 7 6 7 8
1 3 3 4 5 6 6 7 6 7 8
1 3 3 4 5 6 6 7 6 7 8
1 3 3 4 5 6 6 6 7 7 8
1 3 3 4 5 6 6 6 7 7 8
```

Partie 4

Structures de données

LISTES CHAÎNÉES 19

19.1 QU'EST-CE QU'UNE LISTE CHAÎNÉE ?

Une liste chaînée (voir figure 19.1) est un ensemble de cellules liées entre elles par des pointeurs. Chaque cellule est une structure contenant les champs suivants :

- une ou plusieurs données comme dans n'importe quelle structure ;
- un pointeur `suivant` sur la cellule suivante.

On accède à la liste par un pointeur `L` sur la première cellule, puis en parcourant la liste d'une cellule à l'autre en suivant les pointeurs `suivant`. Le dernier pointeur suivant vaut `NULL`, ce qui indique la fin de la liste.

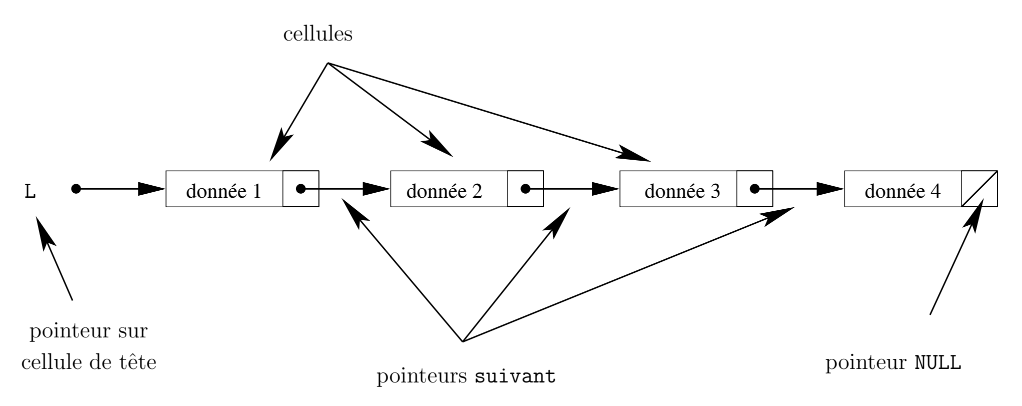


Figure 19.1 – Exemple de liste chaînée avec 4 cellules

19.2 DÉCLARER UNE LISTE CHAÎNÉE

Pour créer une liste chaînée, il faut déclarer une nouvelle structure de données : la structure qui représentera une cellule.

```
/* exemple : les données dans les cellules sont des float */
typedef float TypeDonnee;

/* définition du type cellule :              */

typedef struct Cell
```

```
{
  TypeDonnee donnee;   /* définition des données */
  /* on peut mettre ce qu'on veut comme donnée */

  struct Cell *suivant; /* pointeur sur la structure suivante  */
  /* (de même type que celle qu'on est en train de définir) */
}TypeCellule;
```



La structure TypeCellule contient un pointeur sur TypeCellule. En fait, on ne peut pas déclarer directement un pointeur sur le type TypeCellule qui est en cours de définition. Par contre, le langage *C* permet de définir un pointeur sur une structure non encore définie en employant le mot struct (voir Chapitre 7). De cette manière, le compilateur accepte que le code “se morde la queue”.

On déclare ensuite le pointeur qui donne l'adresse de la première cellule (NULL si la liste est vide) :

```
TypeCellule *L; /* déclaration d'une liste */
```

Ici, nous avons déclaré une liste chaînée de float. Nous aurions pu déclarer une liste chaînée d'entiers, ou d'autres types de données, ou même contenant plusieurs types de données. En fait, la cellule n'est rien d'autre qu'une structure *C* dans laquelle on met des données dans des champs. Ici, nous avons défini un type TypeDonnee par un typedef. Dans la suite, nous écrirons le code des fonctions en utilisant l'identificateur TypeDonnee. Cela permet d'adapter facilement le code pour d'autres types de données (int, char*, int*, structure...) en changeant simplement la définition de TypeDonnee au niveau du typedef et les fonctions d'entrée-sortie (saisie et affichage). On parle d'un code *générique* pour parler d'un code qui peut fonctionner pour différents types de données.
Voici les fonctions d'entrée-sortie de TypeDonnée (dont l'implémentation dépend du type) :

```
void AfficheDonnee(TypeDonnee donnee)
{
  printf("%f ", donnee); /* ici donnée est de type float */
}

TypeDonnee SaisieDonnee(void)
{
  TypeDonnee donnee;
  scanf("%f", &donnee); /* ici donnée est de type float */
  return donnee;
}
```

19.3 Insertion en tête de liste

La fonction suivante prend en paramètre une liste et une donnée, et ajoute la donnée en tête de liste. La fonction renvoie la nouvelle adresse de la tête de liste (voir figure 19.2).

```
TypeCellule* InsereEnTete(TypeCellule *ancienL,
                          TypeDonnee donnee)
{
  TypeCellule *nouveauL;  /* nouvelle tête de liste */
  /* création d'une nouvelle cellule */
  nouveauL = (TypeCellule*)malloc(sizeof(TypeCellule));
  nouveauL->donnee=donnee;    /* on met la donnée à ajouter */
                              /* dans la cellule */
  nouveauL->suivant=ancienL;  /* chaînage */
  return nouveauL; /* on retourne la nouvelle tête de liste */
}
```

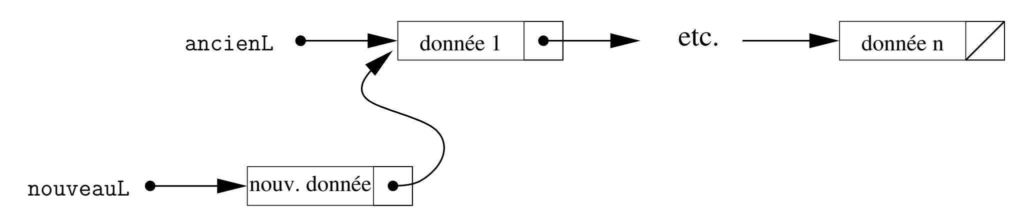


Figure 19.2 – **Insertion en tête de liste**



Ne pas confondre l'utilisation du point `.` et l'utilisation de la flèche `->` pour accéder aux champs d'une structure. On utilise le point pour une variable de type structure, et une flèche pour une variable de type **pointeur** sur structure.

19.4 Construction d'une liste chaînée

Les listes chaînées se construisent par des insertions sucessives. La fonction suivante réalise la saisie d'une liste chaînée au clavier.

```
TypeCellule* SaisieListeEnvers()
{
  char choix;
  TypeDonnee donnee;
  /* déclaration d'une liste vide : */
  TypeCellule *L=NULL;  /*  initialisation obligatoire ! */
  puts("Voulez-vous entrer une liste non vide ?");
```

```
    choix = getchar();
    getchar();
    while (choix == 'o')
      {
        puts("Entrez une donnée");
        donnee = SaisieDonnee(); /* saisie au clavier */
        getchar();
        L = InsereEnTete(L, donnee);  /* insertion en tête */
        /* ne pas oublier de récupérer la nouvelle tête dans L */
        puts("Voulez-vous continuer ?");
        choix = getchar();
        getchar();
      }
    return L;
  }
```

Remarque
Avec des insertions en tête de liste, la liste obtenue est classée à l'envers, le dernier élément saisi étant le premier élément de la liste. Pour obtenir les éléments à l'endroit, il faurait faire les insertions en queue de liste (voir la section 19.6).

Remarque
Nous avons fait une fonction `InsereEnTete` qui retourne la nouvelle tête de liste (type de retour `TypeCellule*`) pour éviter de faire un passage de pointeur par adresse. En effet, nous voulons modifier l'adresse de la tête de liste, donc modifier la valeur du pointeur `L`. Nous aurions pu, en utilisant un double pointeur (`TypeCellule**`) modifier cette valeur par un passage du pointeur par adresse (passage de `&L`). Nous avons jugé plus simple de faire retourner la nouvelle valeur par la fonction. Pour un exemple de passage de pointeur par adresse, voir la section 19.7.

19.5 Parcours de liste

L'idée du parcours de liste chaînée est de prendre un pointeur auxiliaire p. On fait pointer p sur la première cellule, puis le pointeur p passe à la cellule suivante (par une affectation `p=p->suivant`), etc. (voir la figure 19.3). Le parcours s'arrête lorsque p vaut le suivant de la dernière cellule, c'est-à-dire lorsque p vaut `NULL`.

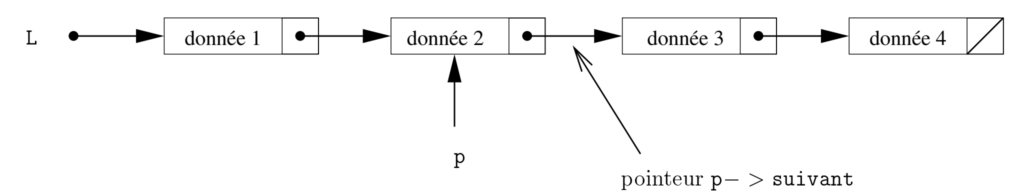


Figure 19.3 – **Parcours de liste chaînée**

Exemple

La fonction suivante réalise l'affichage d'une liste chaînée.

```
void Affichage(TypeCellule* L)
{
  TypeCellule *p;
  p = L;   /* on pointe sur la première cellule */
  while (p != NULL)   /* tant qu'il y a une cellule */
    {
      AfficheDonnee(p->donnee);  /* on affiche la donnée */
      p = p->suivant;  /* on passe à la cellule suivante */
    }
  puts("");  /* passage à la ligne */
}
```



Lors du parcours, on s'arrête lorsque `p` vaut `NULL`, et non pas lorsque `p->suivant` vaut `NULL`. En effet, `p->suivant` vaut `NULL` lorsque `p` pointe sur la dernière cellule (voir la figure 19.4). Il faut traiter cette dernière.

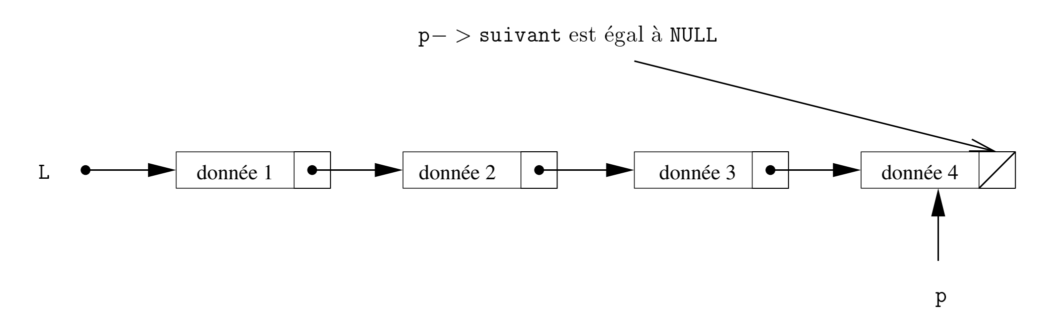


Figure 19.4 - Le pointeur `p` pointe sur la dernière cellule

On pourra par exemple avoir le programme principal suivant :

```
int main(void)
{
  /* déclaration du pointeur sur tête de liste : */
  TypeCellule *L;
  L = SaisieListeEnvers(); /* on récupère l'adresse */
                           /* de la première cellule */
  Affichage(L); /* on affiche la liste saisie */
  return 0;
}
```

On peut aussi faire le parcours de liste chaînée avec une boucle `for` :

```
void AffichageBis(TypeCellule* L)
{
  TypeCellule *p;
  /* tant qu'il y a une cellule */
  for (p=L ; p!=NULL ; p=p->suivant)
      AfficheDonnee(p->donnee);  /* on affiche la donnée */
  puts("");  /* passage à la ligne */
}
```

19.6 Insertion en queue de liste

L'ajout d'une cellule en queue de liste est un peu plus compliquée que l'insertion en tête de liste. Notamment, elle nécessite un parcours de la liste pour rechercher l'adresse du dernier élément (voir la figure 19.5).

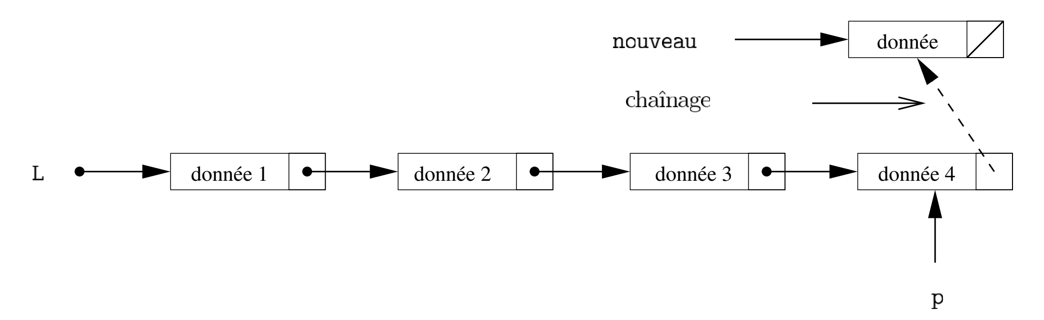


Figure 19.5 – **Insertion en queue d'une liste chaînée**

```
TypeCellule *InsereEnQueue(TypeCellule *L, TypeDonnee donnee)
{
  TypeCellule *p, *nouveau;
  /* allocation d'une nouvelle cellule : */
  nouveau = (TypeCellule*)malloc(sizeof(TypeCellule));
  nouveau->donnee = donnee;  /* donnée de la nouvelle cellule */
  nouveau->suivant = NULL;  /* la nouvelle dernière cellule */
  if (L == NULL)  /* cas particulier si la liste est vide */
    L = nouveau;
  else
    {
      /* recherche de la dernière cellule */
      for (p = L ; p->suivant!=NULL ; p=p->suivant)
        {}
```

```
      p->suivant = nouveau;  /* chaînage */
    }
  return L;
}
```

Ici, on a une condition d'arrêt p->suivant !=NULL parce que nous cherchons l'adresse de la dernière cellule. C'est un cas différent du parcours de liste chaînée, où il faut traîter la dernière cellule comme les autres (condition p !=NULL).
L'insertion en queue de liste permet de saisir une liste chaînée à l'endroit :

```
TypeCellule* SaisieListeEndroit()
{
  char choix;
  TypeDonnee donnee;
  /* déclaration d'une liste vide : */
  TypeCellule *L=NULL;  /*  initialisation obligatoire ! */
  puts("Voulez-vous entrer une liste non vide ?");
  choix = getchar();
  getchar();
  while (choix == 'o')
    {
      puts("Entrez une donnée");
      donnee = SaisieDonnee();
      getchar();
      L = InsereEnQueue(L, donnee);  /* insertion en queue */
      puts("Voulez-vous continuer ?");
      choix = getchar();
      getchar();
    }
  return L;
}
```


Compléments

√ La création d'une liste chaînée par insertions en queue de liste telle que décrite ici prend un nombre d'opérations quadratique ($O(n^2)$) par rapport au nombre de cellules. On peut éviter cela et écrire un algorithme linéaire ($O(n)$) en maintenant tout au long de l'algorithme un pointeur sur la dernière cellule, sans le rechercher à chaque fois par un parcours de liste.

19.7 Libération de mémoire

Pour libérer la mémoire d'une liste chaînée, il faut **détruire chacune des cellules** avec la fonction free. Pour éviter les éventuels bugs, il vaut mieux que la fonction

réinitialise la tête de liste à NULL (liste vide). Pour cela, la fonction doit modifier le pointeur de tête de liste, et il faut donc passer ce pointeur par adresse.

```
void Liberation(TypeCellule **pL)
     /* passage d'un pointeur par adresse : */
     /* pointeur sur pointeur               */
{

  TypeCellule *p;
  while (*pL != NULL)  /* tant que la liste est non vide */
    {
      p = *pL;   /* mémorisation de l'adresse de la cellule */
      *pL = (*pL)->suivant; /* cellule suivante */
      free(p); /* destruction de la cellule mémorisée */
    }
  *pL = NULL; /* on réinitialise la liste à vide */
}
```

```
int main(void)
{
  /* déclaration du pointeur sur tête de liste : */
  TypeCellule *L;
  L = SaisieListeEndroit(); /* on récupère l'adresse */
                            /* de la première cellule */
  Affichage(L); /* on affiche la liste saisie */

  Liberation(&L); /* passage de l'adresse du pointeur */
  return 0;
}
```

Après l'appel de la fonction Liberation, la liste est vide (pointeur NULL).

Exercices

19.1 (∗) Écrire une fonction qui calcule la somme des éléments d'une liste chaînée d'entiers.

19.2 (∗) (recherche d'un élément) Écrire une fonction qui prend en paramètre une liste chaînée d'entiers et un nombre entier n, et qui renvoie l'adresse de la première cellule dont la donnée vaut n. La fonction renverra NULL si l'élément n n'est pas présent dans la liste.

19.3 (∗)

a) Écrire une fonction qui prend en paramètre un tableau et son nombre d'éléments, et qui crée une liste chaînée dont les éléments sont les mêmes que les éléments du tableau.

b) Écrire une fonction qui prend en paramètre une liste chaînée, et qui crée un tableau dont les éléments sont les mêmes que les éléments du tableau.

19.4 (∗) Écrire une fonction qui prend en paramètre une liste chaînée et renvoie une autre liste ayant les mêmes éléments, mais dans l'ordre inverse.

19.5 (∗) Écrire une fonction de recopie d'une liste chaînée.

19.6 (∗) **(suppression dans une liste chaînée)**

a) Écrire une fonction qui prend en paramètre une liste chaînée et une donnée, et qui supprime la première occurrence de cette donnée dans la liste.

b) Écrire une fonction qui supprime toutes les occurrences d'une donnée (passée en paramètre) dans une liste chaînée. Quelle est la complexité de l'algorithme ?

c) Même question qu'au *b*) mais pour un tableau au lieu d'une liste chaînée.

19.7 (∗) Écrire une fonction de concaténation de deux listes chaînées. La fonction prend en entrée deux listes et ressort une seule liste réunion l'une à la suite de l'autre des deux listes. On donnera deux versions de cette fonction, l'une destructrice (c'est-à-dire qu'on ne conservera pas les listes chaînées d'entrée), l'autre non destructrice (c'est-à-dire que l'on préservera les listes chaînées d'entrée).

19.8 (∗∗) **(tri d'une liste chaînée)**

a) Écrire une fonction qui prend en paramètre une liste chaînée d'entiers et vérifie qu'elle est triée dans l'ordre croissant.

b) Écrire une fonction qui insère un élément dans une liste chaînée triée. La liste retournée doit être triée. Pour cela, on recherchera l'adresse de la cellule (si elle existe) juste avant l'emplacement de l'insertion.

c) Même question qu'au b) mais avec un tableau au lieu d'une liste chaînée.

d) Donner un algorithme quadratique de tri d'une liste chaînée (on calculera le nombre d'opérations).

19.9 (∗) Écrire une fonction qui prend en entrée deux listes chaînées de même longueur, et qui crée une liste chaînée fusion alternée des deux listes. La liste fusion doit être la réunion des deux listes, mais les éléments de la liste fusion doivent être alternativement de l'une et de l'autre des deux listes. Si les listes ne sont pas de même longueur, la liste fusion se terminera comme la plus longue des deux listes en entrée. On fera une version destructrice et une version non destructrice de la fonction.

19.10 (∗∗) **(interclassement et tri par interclassement)**

a) Écrire une fonction `Interclassement` qui prend en paramètre deux listes chaînées supposées triées, et qui retourne une liste qui est la réunion des deux listes triées. La liste retournée doit être triée. Quelle est la complexité de l'interclassement ?

b) Écrire une fonction de tri de liste chaînée `TriInterclassement` qui fonctionne comme suit ;

- Si la liste est vide ou n'a qu'un seul élément, la fonction renvoie la liste elle même.
- Sinon, la fonction partage la liste en deux sous-listes `L1` et `L2` d'égales longueurs (plus ou moins 1) et
  1. trie les listes `L1` et `L2` par un appel récursif à la fonction `TriInterclassement` ;
  2. effectue l'interclassement de `L1` et `L2` et retourne le résultat.

Quelle est la complexité de ce tri ?

c) Même question qu'au *a*) mais avec des tableaux au lieu de listes chaînées. La fonction retournera un tableau.

d) Même question qu'au *b*) mais avec des tableaux au lieu des listes chaînées.

19.11 (∗) Écrire une fonction qui prend en entrée une liste chaînée d'entiers et qui ressort deux listes chaînées, l'une avec les nombres pairs, l'autre avec les nombres impairs de la liste d'entrée. On détruira la liste d'entrée.

19.12 (∗∗) **(Listes doublement chaînées)** On s'intéresse à des listes *doublement chaînées*, pour lesquelles chaque cellule à un pointeur `suiv` vers la cellule suivante, et un pointeur `preced` vers la cellule précédente. Voici une exemple ci-dessous :

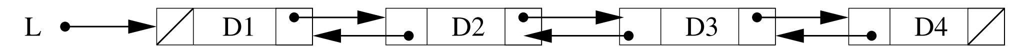


Figure 19.6 – Une liste doublement chaînée

a) Déclarer le type de données correspondant à une liste doublement chaînée d'entiers.

b) Écrire une fonction d'insertion en tête de liste dans une liste doublement chaînée.

c) Écrire une fonction d'insertion en queue de liste dans une liste doublement chaînée.

d) Écrire une fonction d'insertion dans une liste triée doublement chaînée. La liste doit rester triée.

e) Écrire une fonction qui supprimme la première occurence (si elle existe) d'un entier *n* dans une liste doublement chaînée.

f) Écrire une fonction qui supprimme la dernière occurence (si elle existe) d'un entier *n* dans une liste doublement chaînée.

g) Écrire une fonction qui supprimme l'avant-dernière occurence (si elle existe) d'un entier *n* dans une liste doublement chaînée.

h) Écrire une fonction qui supprime toutes les occurences d'un nombre *n* dans une liste doublement chaînée.

i) Écrire une fonction non conservative qui réalise la concaténation de deux listes doublement chaînées.

j) Écrire une fonction de recopie d'une liste doublement chaînée à l'identique.

Corrigés

19.1

```
int Somme(TypeCellule * L)
{
  TypeCellule *p;
  int somme = 0;
  p = L;
  while (p != NULL)
    {
      somme += p->donnee;
      p = p->suivant;
    }
  return somme;
}
```

19.2

```
TypeCellule *Recherche(TypeCellule * L, TypeDonnee n)
{
  TypeCellule *p;
```

```
  p = L;
  while (p != NULL && p->donnee != n)
    p = p->suivant;
  if (p == NULL)
    return NULL;
  return p;
}
```

19.3

a)

```
TypeCellule *CreerListeduTableau(TypeDonnee * tab, int n)
{
  int i;
  TypeDonnee donnee;
  TypeCellule *nouvelleListe = NULL;
  for (i = 0; i < n; i++)
    nouvelleListe = InsereEnQueue(nouvelleListe, tab[i]);
  return nouvelleListe;
}
```

b)

```
TypeDonnee *CreerTableaudelaListe(TypeCellule * L, int *n)
{
  int i = 0;
  TypeDonnee *tab;
  TypeCellule *p;
  p = L;
  *n = 0;        /* compter le nb d'éléments à partir de zéro */
/* compter le nombre d'éléments dans la liste */
  while (p != NULL)
    {
      (*n)++;
      p = p->suivant;
    }
  tab = (TypeDonnee *) malloc((*n) * sizeof(TypeDonnee));
  p = L;        /* retourner au début de la liste */
  while (p != NULL)        /* remplir le tableau */
    {
      tab[i] = p->donnee;
      p = p->suivant;
      i = i + 1;
    }
  return tab;
}
```

19.4

```
TypeCellule *CreerListeEnvers(TypeCellule * L)
{
  TypeCellule *p, *nouveauL = NULL;
  p = L;
  while (p != NULL)
    {
      nouveauL = InsereEnTete(nouveauL, p->donnee);
      p = p->suivant;
    }
  return nouveauL;
}
```

19.5

```
TypeCellule *RecopieListe(TypeCellule * L)
{
  TypeCellule *p, *nouveauL = NULL;
  p = L;
  while (p != NULL)
    {
      nouveauL = InsereEnQueue(nouveauL, p->donnee);
      p = p->suivant;
    }
  return nouveauL;
}
```

19.6

a)

```
TypeCellule *Supprime1occur(TypeCellule * L, TypeDonnee donnee)
{
  TypeCellule *p, *suivant, *pL;
  p = L;
  if (p == NULL)
    return L;
/*Si la première occurrence est le premier élément de L*/
  if (p->donnee == donnee)
    {
      pL = p->suivant;
      free(p);
      return pL;
    }
/*Pour les autres cas, on maintient 2 pointeurs: un vers
la cellule courante et un vers la cellule suivante*/
  suivant = p->suivant;
```

```
  while (suivant->suivant != NULL && suivant->donnee != donnee)
    {
      p = p->suivant;
      suivant = p->suivant;
    }
  if (suivant->donnee == donnee)
    {
      free(suivant);
      p->suivant = suivant->suivant;
    }
  return L;
}
```

b)

```
TypeCellule *Supprimetouteoccur(TypeCellule * L, TypeDonnee donnee)
{
  TypeCellule *p, *suivant, *pL;
  p = L;
  if (p == NULL)
    return L;
  while (p->suivant != NULL && p->donnee == donnee)
    {
      pL = p;
      p = p->suivant;
      free(pL);
    }
  if (p->donnee == donnee)
    {
      pL = p;
      L = p->suivant;
      free(pL);
    }
/*Pour les autres cas, on maintient 2 pointeurs: un vers
 la cellule courante et un vers la cellule suivante*/
  suivant = p->suivant;
  while (suivant->suivant != NULL)
    {
      if (suivant->donnee == donnee)
        {
          p->suivant = suivant->suivant;
          free(suivant);
        }
      else
        p = p->suivant;
```

```
        suivant = p->suivant;
      }
    if (suivant->donnee == donnee)
      {
        p->suivant = NULL;
        free(suivant);
      }
    return L;
}
```

La complexité est $O(n)$

c)

```
void Supprimetouteoccurtab(TypeDonnee * tab, TypeDonnee donnee, int *nb)
{
  int i = 0, j = 0;
  while (i < *nb)
    {
      if (tab[i] == donnee)
        i++;
      else
        {
          tab[j] = tab[i];
          i++;
          j++;
        }
    }
  *nb = j;
}
```

La complexité est de $O(n)$

19.7

```
TypeCellule *Concatenationdestructrice(TypeCellule * L1,
                                       TypeCellule * L2)
{
  TypeCellule *p;
  p = L1;
  while (p->suivant != NULL)
    {
      p = p->suivant;
    }
  p->suivant = L2;
  return L1;
}
```

```
TypeCellule *Concatenationnondestructrice(TypeCellule * L1,
                                          TypeCellule * L2)
{
  TypeDonnee donnee;
  TypeCellule *p, *L3 = NULL;
  p = L1;
  while (p != NULL)
    {
      L3 = InsereEnQueue(L3, p->donnee);
      p = p->suivant;
    }
  p = L2;
  while (p != NULL)
    {
      L3 = InsereEnQueue(L3, p->donnee);
      p = p->suivant;
    }
  return L3;
}
```

19.8

a)

```
/*la fonction retourne 0 si la liste est triée,
-1 si elle est vide et 1 si elle n'est pas triée */
int Verifietrie(TypeCellule * L)
{
  TypeCellule *p, *suivant;
  if (L == NULL)
    {
      puts("La liste est vide");
      return -1;
    }
  p = L;
  suivant = p->suivant;
  if (suivant == NULL) /* la liste contient un seul élément */
    return 0;
  while (suivant->suivant != NULL && p->donnee < suivant->donnee)
    {
      p = p->suivant;
      suivant = p->suivant;
    }
  if (suivant->suivant == NULL) /* on arrive au dernier élément */
```

```
    {
      if (p->donnee < suivant->donnee)
        return 0;        /*la liste est triée */
      else
        return 1;
    }
  return 1;
}
```

b)

```
TypeCellule *InsereElement(TypeCellule * L, TypeDonnee donnee)
{
  TypeCellule *p, *nouveau, *suivant;
  nouveau = (TypeCellule *) malloc(sizeof(TypeCellule));
  nouveau->donnee = donnee;
  nouveau->suivant = NULL;
  if (L == NULL)
    return nouveau;
  p = L;
  suivant = p->suivant;
/*Si la première donnée est < que la donnée à insérer */
  if (p->donnee < donnee)
    {
      if (suivant == NULL)
        {
          p->suivant = nouveau;
          return L;
        }
    }
  else        /* Si la première donnée est > à la donnée à insérer */
    {
      nouveau->suivant = p;
      return nouveau;
    }
  while (suivant->suivant != NULL && suivant->donnee < donnee)
    {
      p = p->suivant;
      suivant = p->suivant;
    }
  if (suivant->donnee > donnee)
    {
      p->suivant = nouveau;
      nouveau->suivant = suivant;
    }
```

```
  else
    suivant->suivant = nouveau;
  return L;
}
```

c)

```
void InsereElementTabTriee(TypeDonnee * tab, TypeDonnee donnee, int *nb)
{
  int i = 0, j = 0;
  while (i < *nb && tab[i] < donnee)
    i++;
/*Augmenter de 1 la taille du tableau alloué */
  tab = realloc(tab, (*nb + 1) * sizeof(int));
  if (i == *nb)
    tab[i] = donnee;
  else
    {
      for (j = (*nb) + 1; j > i; j--)
        tab[j] = tab[j - 1];
/*insérer la donnée à son emplacement i */
      tab[i] = donnee;
    }
  *nb = *nb + 1;
}
```

d)

```
TypeCellule *TriListe(TypeCellule * L)
{
  TypeCellule *p, *Listetriee = NULL;
  p = L;
  while (p != NULL)
    {
      Listetriee = InsereElement(Listetriee, p->donnee);
      p = p->suivant;
    }
  return Listetriee;
}
```

19.9

```
/*Fusion destructrice*/
TypeCellule *FusionDestructrice(TypeCellule * L1, TypeCellule * L2)
```

```
{
  TypeCellule *p1, *p2, *suivant_p1, *suivant_p2;
  p1 = L1;
  p2 = L2;
  while (p1->suivant != NULL && p2->suivant != NULL)
    {
/*mémoriser le suivant de p1 */
      suivant_p1 = p1->suivant;
/*mémoriser le suivant de p2 */
      suivant_p2 = p2->suivant;
      p1->suivant = p2;
      p2->suivant = suivant_p1;
      p1 = suivant_p1;
      p2 = suivant_p2;
    }
/* L1 est plus longue que L2 */
  if (p1->suivant != NULL && p2->suivant == NULL)
    {
      suivant_p1 = p1->suivant;
      p1->suivant = p2;
      p2->suivant = suivant_p1;
    }
  /* L1 et L2 ont la même longueur ou L2 est plus longue que L1 */
  else
    p1->suivant = p2;
  return L1;
}

 /*Fusion non destructrice */
TypeCellule *FusionNonDestructrice(TypeCellule * L1, TypeCellule * L2)
{
  TypeCellule *p1, *p2, *L3 = NULL;
  p1 = L1;
  p2 = L2;
/*si la fin d'aucune des listes n'est atteinte */
  while (p1 != NULL && p2 != NULL)
    {
      L3 = InsereEnQueue(L3, p1->donnee);
      L3 = InsereEnQueue(L3, p2->donnee);
      p1 = p1->suivant;
      p2 = p2->suivant;
    }
  /*si L1 est plus longue que L2 et fin de L2 atteinte */
  while (p1 != NULL)
    {
      L3 = InsereEnQueue(L3, p1->donnee);
```

```
      p1 = p1->suivant;
    }
/*L2 est plus longue que L1 et la fin de L1 est atteinte */
  while (p2 != NULL)
    {
      L3 = InsereEnQueue(L3, p2->donnee);
      p2 = p2->suivant;
    }
  return L3;        /*retourner la nouvelle liste */
}
```

19.10

a)

```
TypeCellule *Interclassement(TypeCellule * Liste1, TypeCellule * Liste2)
{
  TypeCellule *tmp, *debut;
  if (Liste1->donnee <= Liste2->donnee)
    {
      tmp = Liste1;
      debut = Liste1;
      Liste1 = Liste1->suivant;
    }
  else
    {
      tmp = Liste2;
      debut = Liste2;
      Liste2 = Liste2->suivant;
    }
  while (tmp->suivant != NULL)
    {
      if (Liste1->donnee <= Liste2->donnee)
        {
          tmp->suivant = Liste1;
          tmp = Liste1;
          Liste1 = Liste1->suivant;
        }
      else
        {
          tmp->suivant = Liste2;
          tmp = Liste2;
          Liste2 = Liste2->suivant;
        }
    }
```

```
  if (Liste1 != NULL)
    tmp->suivant = Liste1;
  if (Liste2 != NULL)
    tmp->suivant = Liste2;
  return debut;
}
```

b)

```
TypeCellule *TriInterclassement(TypeCellule * Liste)
{
  TypeCellule *debutliste, *tmp, *res1, *res2;
  if (Liste == NULL || Liste->suivant == NULL)
    return Liste;
  debutliste = Liste;
  tmp = Liste->suivant->suivant;
  while (tmp != NULL && tmp->suivant != NULL)
    {
      tmp = tmp->suivant->suivant;
      Liste = Liste->suivant;
    }
  tmp = Liste->suivant;
  Liste->suivant = NULL;
  res1 = TriInterclassement(debutliste);
  res2 = TriInterclassement(tmp);
  return Interclassement(res1, res2);
}
```

c)

```
void Interclassement(TypeDonnee * tab, TypeDonnee * tmptab,
                int debut, int m, int fin)
{
  int i, j, k;
  for (i = m + 1; i > debut; i--)
    tmptab[i - 1] = tab[i - 1];
  for (j = m; j < fin; j++)
    tmptab[fin + m - j] = tab[j + 1];
  for (k = debut, i = debut, j = fin; k <= fin; k++)
    tab[k] = (tmptab[i] < tmptab[j]) ? tmptab[i++] : tmptab[j--];
}
```

d)

```
void TriInterclassement(TypeDonnee * tab, TypeDonnee * tmptab,
                         int debut, int fin)
{
  int m;
```

```
  int i, j, k;
  if (fin > debut)
    {
      m = (debut + fin) / 2;
      TriInterclassement(tab, tmptab, debut, m);
      TriInterclassement(tab, tmptab, m + 1, fin);
      Interclassement(tab, tmptab, debut, m, fin);
    }
}
```

La complexité est $O(nlog_2 n)$

19.11

```
void Pairimpair(TypeCellule * L, TypeCellule ** L1, TypeCellule ** L2)
{
  TypeCellule *p, *p1, *p2;
  *L1 = NULL;
  *L2 = NULL;
  p = L;
  while (p->suivant != NULL)
    {
      if ((p->donnee) % 2 == 0)
        {
          if (*L1 == NULL)
            {
              *L1 = p;
              p1 = *L1;
            }
          else
            {
              p1->suivant = p;
              p1 = p1->suivant;
            }
        }
      else
        {
          if (*L2 == NULL)
            {
              *L2 = p;
              p2 = *L2;
            }
          else
            {
              p2->suivant = p;
```

```
                p2 = p2->suivant;
              }
          }
        p = p->suivant;
    }
}
```

19.12

a)

```
typedef struct Cell
{
  TypeDonnee donnee;
  struct Cell * precedent;
  struct Cell * suivant;
} TypeCellule;
```

b)

```
TypeCellule* InsereEnTete(TypeCellule *ancienL, TypeDonnee donnee)
{
  TypeCellule *nouveauL;
  nouveauL=(TypeCellule*)malloc(sizeof(TypeCellule));
  nouveauL->donnee=donnee;
  nouveauL->suivant=ancienL;
  nouveauL->precedent=NULL;
  if (ancienL!=NULL)
    ancienL->precedent=nouveauL;
  return nouveauL;
}
```

c)

```
TypeCellule *InsereEnQueue(TypeCellule *L,TypeDonnee donnee)
{
  TypeCellule *p,*nouveau;
  nouveau=(TypeCellule*)malloc(sizeof(TypeCellule));
  nouveau->donnee=donnee;
  nouveau->suivant=NULL;
  if (L==NULL)
    {
      nouveau->precedent=NULL;
      return nouveau;
    }
  else
    {
      for(p=L; p->suivant!=NULL; p=p->suivant)
```

```
        {}
      p->suivant=nouveau;
      nouveau->precedent=p;
    }
  return L;
}
```

d)

```
TypeCellule *InsereDonnee(TypeCellule *L,TypeDonnee donnee)
{
  TypeCellule *precedent,*nouveau,*suivant;
  nouveau=(TypeCellule*)malloc(sizeof(TypeCellule));
  nouveau->donnee=donnee;
  if (L==NULL) /* si la liste est vide */
    {
      nouveau->suivant=NULL;
      nouveau->precedent=NULL;
      return L;
    }
  else if (L->donnee>donnee) /* si l'élément à insérer est */
                         /*le plus petit élément de la liste */
                        /* et doit donc être insérer en tête */
    {
      nouveau->precedent=NULL;
      nouveau->suivant=L;
      L->precedent=nouveau;
      return nouveau;
    }
  else /* dans tous les autres cas */
    {
      precedent=L; /* precedent est celui qui precède le suivant */
      suivant=precedent->suivant;
      while(suivant!=NULL && suivant->donnee<donnee)
        /* tant que le suivant est plus petit que l'élément à */
        /* insérer et tant qu'on n'est pas à la fin de la liste */
        {
          precedent=suivant;
          suivant=suivant->suivant;
        }
      if (suivant==NULL) /*si l'élément à insérer est le dernier */
        {
          precedent->suivant=nouveau;
          nouveau->precedent=precedent;
          nouveau->suivant=NULL;
        }
```

```
      else
        {
          precedent->suivant=nouveau;
          nouveau->precedent=precedent;
          nouveau->suivant=suivant;
          suivant->precedent=nouveau;
        }
      return L;
    }
}
```

e)

```
TypeCellule *Supprime1occurence(TypeCellule *L,TypeDonnee n)
{
  TypeCellule *precedent,*suivant,*p;
  p=L;
  if (p==NULL) /* si la liste est vide */
    return L;
  /* si la liste n'est pas vide */
  while(p!=NULL && p->donnee!=n) /* tant qu'on n'est pas à l'élément */
    /* n recherché et tant qu'on est pas à la fin de la liste */
    {
      p=p->suivant; /* on avance dans la liste */
    }
  if (p!=NULL) /* si on a trouvé n */
    {
      precedent=p->precedent;
      suivant=p->suivant;
      free(p);
      if (suivant!=NULL) /* si le suivant existe */
        suivant->precedent=precedent;
      if (precedent!=NULL)
        precedent->suivant=suivant;
      else /* precedent==NULL */
        return suivant;
    }
  return L;
}
```

f)

```
TypeCellule *SupprimeDOccurence(TypeCellule *L,TypeDonnee n)
{
  TypeCellule *precedent,*suivant,*p;
  p=L;
```

```
  if (p==NULL) /* si la liste est vide */
    return L;

  /* si la liste n'est pas vide */
  while(p->suivant!=NULL) /* on avance jusqu'au dernier élement */
    p=p->suivant;

  /* on parcours la liste à partir de la fin jusqu'au début
     afin de supprimer la dernière occurence de n */
  while(p!=NULL && p->donnee!=n) /* tant qu'on est pas au élément */
      /* n recherché et tant qu'on est pas au début de la liste */
    {
      p=p->precedent; /* on avance dans la liste */
    }
  if (p!=NULL) /* si on a trouvé n */
    {
      precedent=p->precedent;
      suivant=p->suivant;
      free(p);
      if (suivant!=NULL) /* si le suivant existe */
        suivant->precedent=precedent;
      if (precedent!=NULL)
        precedent->suivant=suivant;
      else /* precedent==NULL */
        return suivant; /* le premier élément de la liste a changé */
    }
  return L;
}
```

g)

```
TypeCellule *SupprimeAvantDerniere(TypeCellule *L,TypeDonnee n)
{
  TypeCellule *precedent,*suivant,*p;
  int compteur=0; /* sert à compter les occurence de n. */
                  /*  il sera égal à 2 à l'avant dernière */
                  /* occurence de n en parcourant à l'envers */
  p=L;
  if (p==NULL) /* si la liste est vide */
    return L;
  /* si la liste n'est pas vide */
  while(p->suivant!=NULL) /* on avance jusqu'au dernier élement */
    p=p->suivant;
  /* si la liste n'est pas vide, on parcourt à partir de la fin*/
  while(p!=NULL) /* tant qu'on est pas à la fin de la liste */
```

```
    {
      if (p->donnee==n && compteur==0)
      /* si on arrive à un élément égal à n avant compteur=2 */
        compteur++;
      else if (p->donnee==n && compteur==1) /* si on arrive au 2ème
                                               n à partir de la fin*/
        {
          precedent=p->precedent;
          suivant=p->suivant;
          free(p);
          if (suivant!=NULL) /* si le suivant existe */
            suivant->precedent=precedent;
          if (precedent!=NULL)
            precedent->suivant=suivant;
          else /* precedent==NULL */
            return suivant; /* le premier élément a changé */
          return L;
        }
      p=p->precedent; /* avancer au suivant */
    }
  return L; /* l'avant dernière occurence n'existe pas */
}
```

h)

```
TypeCellule *SupprimeTouteOccurence(TypeCellule *L,TypeDonnee n)
{
  TypeCellule *precedent,*suivant,*p,*pfree;
  p=L;
  if (p==NULL) /* si la liste est vide */
    return L;
  /* si la liste n'est pas vide */
  while(p!=NULL) /* tant qu'on est pas à la fin de la liste */
    {
      if (p->donnee==n) /* si on arrive à un élément égal à n */
        {
          precedent=p->precedent;
          suivant=p->suivant;
          if (suivant!=NULL) /* si le suivant existe */
            suivant->precedent=precedent;
          if (precedent!=NULL)
            precedent->suivant=suivant;
          else /* precedent==NULL */
            L=suivant;
          pfree=p; /* on mémorise la cellule à supprimer */
```

```
          free(pfree);
          p=suivant; /* avancer au suivant */
        }
      else
        p=p->suivant; /* avancer au suivant */
    }
  return L;
}
```

i)

```
TypeCellule* ConcatenatDestruct(TypeCellule *L1, TypeCellule *L2)
{
  TypeCellule *p;
  p=L1;
  if (L1==NULL) /* si L1 est vide */
    return L2;
  if (L2==NULL) /* si L2 est vide */
    return L1;

  while(p->suivant!=NULL) /* avancer jusque la fin de L1 */
    p=p->suivant;

  p->suivant=L2;
  L2->precedent=p;
  return L1;
}
```

j)

```
TypeCellule * RecopieListe (TypeCellule * L)
{
  TypeCellule *p, *nouveauL = NULL;
  p = L;
  while (p != NULL)
    {
      nouveauL = InsereEnQueue (nouveauL, p->donnee);
      p = p->suivant;
    }
  return nouveauL;
}
```

PILES 20

20.1 QU'EST-CE QU'UNE PILE ?

Une *pile* est une structure de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du *dernier arrivé premier sorti* ou encore *LIFO* (*Last In First Out*).

Le nom de pile vient d'une analogie avec une pile d'assiettes (par exemple) où l'on poserait toujours les assiettes sur le dessus de la pile, et ou l'on prendrait toujours les assiettes sur le dessus de la pile. Ainsi, la dernière assiette posée sera utilisée avant toutes les autres.

Une pile peut être implémentée par un tableau ou par une liste chaînée. Dans les deux cas, il est commode de réaliser sur les piles des opérations de base, appelées *primitives de gestion des piles*.
Les primitives de gestion des piles sont les suivantes :

- `Initialiser` : cette fonction crée une pile vide.
- `EstVide` : renvoie 1 si la pile est vide, 0 sinon.
- `EstPleine` : renvoie 1 si la pile est pleine, 0 sinon.
- `AccederSommet` : cette fonction permet l'accès à l'information contenue dans le sommet de la pile.
- `Empiler` : cette fonction permet d'ajouter un élément au sommet de la pile. La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.
- `Depiler` : cette fonction supprime le sommet de la pile. L'élément supprimé est retourné par la fonction `Depiler` pour pouvoir être utilisé.
- `Vider` : cette fonction vide la pile.
- `Detruire` : cette fonction permet de détruire la pile.

Le principe de gestion des piles est que, lorsqu'on utilise une pile, on ne se préoccupe pas de la manière dont elle a été implémentée, mais on utilise uniquement les primitives qui sont toujours les mêmes. Nous allons toutefois étudier l'implémentation des primitives de gestion des piles.

20.2 IMPLÉMENTATION SOUS FORME DE TABLEAU

20.2.1 Types

Pour implémenter une pile sous forme de tableau, on crée la structure de données suivante. L'implémentation est donnée pour des données de type `float`. Le principe est le même pour n'importe quel type de données.

```
typedef float TypeDonnee;

typedef struct
{
  int nb_elem;  /* nombre d'éléments dans la pile */
  int nb_elem_max;  /* capacité de la pile */
  TypeDonnee *tab;  /* tableau contenant les éléments */
}Pile;
```

20.2.2 Créer une pile vide

La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser(int nb_max)
{
  Pile pilevide;
  pilevide.nb_elem = 0;   /* la pile est vide */
  pilevide.nb_elem_max = nb_max;  /* capacité nb_max */
  /* allocation des éléments : */
  pilevide.tab = (TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
  return pilevide;
}
```

20.2.3 Pile vide, pile pleine

La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie 1 si le nombre d'éléments est égal à 0. La fonction renvoie 0 dans le cas contraire.

```
int EstVide(Pile P)
{
  /* retourne 1 si le nombre d'éléments vaut 0 */
  return (P.nb_elem == 0) ? 1 : 0;
}
```

La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P)
{
  /* retourne 1 si le nombre d'éléments est supérieur */
  /* au nombre d'éléments maximum et 0 sinon */
  return (P.nb_elem >= P.nb_elem_max) ? 1 : 0;
}
```

20.2.4 Accéder au sommet de la pile

Le sommet de la pile est le dernier élément entré, qui est le dernier élément du tableau. La fonction effectue un passage par adresse pour ressortir le résultat. La fonction permettant d'accéder au sommet de la pile, qui renvoie le code d'erreur 1 en cas de liste vide et 0 sinon, est donc la suivante :

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
  if (EstVide(P))
    return 1;  /* on retourne un code d'erreur */
  *pelem =  P.tab[P.nb_elem-1]; /* on renvoie l'élément */
  return 0;
}
```

20.2.5 Ajouter un élément au sommet

Pour modifier le nombre d'élements de la pile, il faut passer la pile par adresse. La fonction `Empiler`, qui renvoie 1 en cas d'erreur et 0 dans le cas contraire, est la suivante :

```
int Empiler(Pile* pP, TypeDonnee elem)
{
  if (EstPleine(*pP))
    return 1;  /* on ne peut pas rajouter d'élément */
  pP->tab[pP->nb_elem] = elem;   /* ajout d'un élément */
  pP->nb_elem++;  /* incrémentation du nombre d'éléments */
  return 0;
}
```

20.2.6 Supprimer un élément

La fonction `Depiler` supprime le sommet de la pile en cas de pile non vide. La fonction renvoie 1 en cas d'erreur (pile vide), et 0 en cas de succès.

```
char Depiler(Pile *pP, TypeDonnee *pelem)
{
  if (EstVide(*pP))
    return 1;  /* on ne peut pas supprimer d'élément */
  *pelem = pP->tab[pP->nb_elem-1];  /* on renvoie le sommet */
  pP->nb_elem--;  /* décrémentation du nombre d'éléments */
  return 0;
}
```

20.2.7 Vider et détruire

```
void Vider(Pile *pP)
{
  pP->nb_elem = 0; /* réinitialisation du nombre d'éléments */
}
```

```
void Detruire(Pile *pP)
{
  if (pP->nb_elem_max != 0)
    free(pP->tab);  /* libération de mémoire */
  pP->nb_elem = 0;
  pP->nb_elem_max = 0; /* pile de taille 0 */
}
```

20.3 Implémentation sous forme de liste chaînée

20.3.1 Types

Pour implémenter une pile sous forme de liste chaînée, on crée la structure de données suivante. L'implémentation est donnée pour des données de type `float`. Le principe est le même pour n'importe quel type de données.

```
typedef float TypeDonnee;

typedef struct Cell
{
  TypeDonnee donnee;
  struct Cell *suivant; /* pointeur sur la cellule suivante */
}TypeCellule;

typedef TypeCellule* Pile;  /* la pile est un pointeur */
                            /* sur la tête de liste */
```

20.3.2 Créer une pile vide

La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser()
{
  return NULL;  /* on retourne une liste vide */
}
```

20.3.3 Pile vide, pile pleine

La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie 1 si la pile est vide, et renvoie 0 dans le cas contraire.

```
int EstVide(Pile P)
{
  /* renvoie 1 si la liste est vide */
  return (P == NULL) ? 1 : 0;
}
```

La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P)
{
  return 0; /* une liste chaînée n'est jamais pleine */
}
```

20.3.4 Accéder au sommet de la pile

Le sommet de la pile est le dernier élément entré qui est la tête de liste. La fonction renvoie 1 en cas de liste vide, 0 sinon.

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
  if (EstVide(P))
    return 1;  /* on retourne un code d'erreur */
  *pelem = P->donnee; /* on renvoie l'élément */
  return 0;
}
```

20.3.5 Ajouter un élément au sommet

La fonction d'ajout d'un élément est une fonction d'insertion en tête de liste (voir chapitre 19).

```
void Empiler(Pile* pP, TypeDonnee elem)
{
  Pile q;
  q = (TypeCellule*)malloc(sizeof(TypeCellule)); /* allocation */
  q->donnee = elem; /* ajout de l'élément à empiler */
  q->suivant = *pP; /* insertion en tête de liste */
  *pP =q; /* mise à jour de la tête de liste */
}
```

20.3.6 Supprimer un élément

La fonction `Depiler` supprime la tête de liste en cas de pile non vide. La fonction renvoie 1 en cas d'erreur, et 0 en cas de succès. La pile est passée par adresse, on a donc un double pointeur.

```
int Depiler(Pile *pP, TypeDonnee *pelem)
{
  Pile q;
  if (EstVide(*pP))
    return 1; /* on ne peut pas supprimer d'élément */
  *pelem = (*pP)->donnee; /* on renvoie l'élément de tête */
  q = *pP; /* mémorisation d'adresse de la première cellule */
  *pP = (*pP)->suivant; /* passage au suivant */
  free(q); /* destruction de la cellule mémorisée */
  return 0;
}
```

20.3.7 Vider et détruire

La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Detruire(Pile *pP)
{
  Pile q;
  while (*pP != NULL) /* parcours de la liste */
    {
      q = *pP;  /* mémorisation de l'adresse */
      /* passage au suivant avant destruction : */
      *pP = (*pP)->suivant;
      free(q); /* destruction de la cellule mémorisée */
    }
  *pP = NULL; /* liste vide */
}
```

```
void Vider(Pile *pP)
{
  Detruire(pP);  /* destruction de la liste */
  *pP = NULL; /* liste vide */
}
```

20.4 Comparaison entre tableaux et listes chaînées

Dans les deux types de gestion des piles, chaque primitive ne prend que quelques opérations (complexité en temps constant). Par contre, la gestion par listes chaînées présente l'énorme avantage que la pile a une capacité virtuellement illimitée (limitée seulement par la capacité de la mémoire centrale), la mémoire étant allouée à mesure des besoins. Au contraire, dans la gestion par tableaux, la mémoire est allouée au départ avec une capacité fixée.

Exercices

20.1 (∗∗) Écrire un algorithme utilisant une pile (implémentée sous forme de liste chaînée) qui affiche une liste chaînée d'entiers à l'envers.

20.2 (∗∗) Un fichier texte peut contenir des parenthèses ( ), des crochets [ ], et des accolades { }. Ces éléments peuvent être imbriqués les uns dans les autres (exemple : $\{a(bc[d])[\{ef\}(g)]\}$) Écrire une fonction qui parcourt le fichier texte et détermine si le fichier est correctement parenthésé, c'est-à-dire si toutes les parenthèses, crochets, etc. sont bien refermés par un caractère du même type, et si les parenthèses, crochets et accolades sont correctement imbriqués. Exemple de fichier incorrect : ({)}.

20.3 (∗∗) Le but de l'exercice est d'implémenter un jeu de bataille automatique. Chaque carte sera représentée une structure contenant un `int` allant de 0 à 13, et un `char` entre 0 et 3 représentant la couleur (pique, carreau, trèfle ou cœur).

On supposera qu'au début du jeu, toutes les 52 cartes sont rangées dans un ordre aléatoire dans une pile. On représentera le jeux de cartes de chaque joueur par une pile.

a) Proposer une structure de données pour représenter le talon et les jeux des joueurs. On supposera qu'il y a 2 joueurs.

b) Écrire une fonction de distribution qui donne la moitié des cartes à chaque joueur.

c) Écrire une fonction permettant de jouer un coup : chaque joueur pose une carte, la plus grosse carte remporte le pli. En cas d'égalité des cartes, chaque joueur repose une carte et ainsi de suite jusqu'à ce qu'un des joueurs remporte le pli.

d) Écrire une fonction qui effectue l'ensemble du jeu et donne le joueur gagnant.

Corrigés

20.1

```
void Affichierpile(Pile P)
{
  Pile q;
  q = P;
  while (q != NULL)
    {
      printf("%d\t", q->donnee);
      q = q->suivant;
    }
  puts("");
}
```

20.2

```
/* Fonction qui retourne -1 en
cas d'erreur d'ouverture du fichier
0 si le fichier est bien parenthésé
et 1 s'il n'est pas */
int CorrectementParentheser(char *nomFichier)
{
  Pile P;
  FILE *fp;
  TypeDonnee c;
  P = Initialiser();
  if ((fp = fopen(nomFichier, "rt")) == NULL)
    {
      puts("Erreur : fichier inexistant ou droits insuffisants");
      return -1;
    }
  while (!feof(fp))
    {
      c = fgetc(fp);
      switch (c)
```

```
{
case '(':
  {
    Empiler(&P, c);
    break;
  }
case '{':
  {
    Empiler(&P, c);
    break;
  }
case '[':
  {
    Empiler(&P, c);
    break;
  }
case ')':
  {
    if (!EstVide(P) && P->donnee == '(')
      Depiler(&P, &c);
    else
      {
        fclose(fp);
      /* Erreur le fichier n'est pas correctement parenthésé */
        return 1;
      }
    break;
  }
case '}':
  {
    if (!EstVide(P) && P->donnee == '{')
      Depiler(&P, &c);
    else
      {
        fclose(fp);
        return 1;
      }
    break;
  }
case ']':
  {
    if (!EstVide(P) && P->donnee == '[')
      Depiler(&P, &c);
    else
```

```
              {
                              fclose(fp);
                return 1;
              }
            break;
          }
        default:
          break;
        }
    }
  fclose(fp);
/* S’il en reste des données dans la pile le fichier n’est
   pas correctement parenthésé */
  if (!EstVide(P))
    return 1;
  else
    return 0;
}
```

20.3

a)

```
typedef int TypeDonnee;
typedef struct Cell
{
  TypeDonnee donnee;
  struct Cell *suivant;
} TypeCellule;
typedef TypeCellule *Pile;
```

b)

```
void distribuer(Pile P, Pile * P1, Pile * P2)
{
  Pile p1;
  *P1 = Initialiser();
  *P2 = Initialiser();
  p1 = P;
  while (p1 != NULL)
    {
      Empiler(P1, p1->donnee);
      p1 = p1->suivant;
      Empiler(P2, p1->donnee);
      p1 = p1->suivant;
    }
}
```

c)

```
void Jouercoup(Pile * P1, Pile * P2, Pile * talon1, Pile * talon2)
{
  TypeDonnee elem1, elem2;
  Pile tmp;
  tmp = Initialiser();
  int flag = 0;
/* le flag reste 0 tant que elem1 et elem2 sont égaux */
  while (flag == 0)
    {
      if (Depiler1erElement(P1, talon1, &elem1))
        exit(1);
      if (Depiler1erElement(P2, talon2, &elem2))
        exit(1);
      if (elem1 == elem2)
        {
          /* mettre elem1 et elem 2 dans tmp */
          Empiler(&tmp, elem1);
          Empiler(&tmp, elem2);
        }
      if (elem1 > elem2)
        {
          Empiler(talon1, elem1);
          Empiler(talon1, elem2);
          /* tant que tmp contient des cartes,
     celles-ci sont transférées au talon1 */
          while (!EstVide(tmp))
            {
              Depiler(&tmp, &elem1);
              Empiler(talon1, elem1);
            }
          flag = 1;        /* sortir de la boucle while */
        }
      else if (elem1 < elem2)
        {
          Empiler(talon2, elem2);
          Empiler(talon2, elem1);
          /* tant que tmp contiennent des cartes,
     celles-ci sont transférées au talon2 */
          while (!EstVide(tmp))
            {
              Depiler(&tmp, &elem1);
              Empiler(talon2, elem1);
            }
```

```
            flag = 1;          /* plus besoin de boucler */
        }
    }
}

/* Les données du talon sont transférées à la pile
 et le 1er élément est dépilé */
void ChangerdeTas(Pile * P, Pile * talon, TypeDonnee * elem)
{
  *P = *talon;
  *talon = Initialiser();
  Depiler(P, elem);
}

/*Cette fonction retourne le 1er élément de la Pile
s'il existe ; si la Pile est vide, elle échange la
Pile et le talon, et retourne le 1er élément ;
si les deux sont vides elle retourne 1 */
int Depiler1erElement(Pile * P, Pile * talon, TypeDonnee * elem)
{
  char sortie;
  sortie = Depiler(P, elem);
  if (sortie && talon != NULL)
    ChangerdeTas(P, talon, elem);
  else if (sortie && talon == NULL)        /*le joueur a perdu */
    return 1;
  return 0;
}
```

d)

```
void JouerBataille(Pile P)
{
  Pile P1, P2, talon1, talon2;
  talon1 = Initialiser();
  talon2 = Initialiser();
  distribuer(P, &P1, &P2);
  while ((P1 != NULL || talon1 != NULL) && (P2 != NULL || talon2 != NULL))
    Jouercoup(&P1, &P2, &talon1, &talon2);
  if (P1 == NULL && talon1 == NULL)
    puts("Le 1er joueur a gagné");
  else
    puts("Le 2ème joueur a gagné");
}
```

FILES 21

21.1 QU'EST-CE QU'UNE FILE ?

Une *file* est une structures de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du *premier arrivé premier sorti*, ou encore *FIFO* (*First In First Out*).

Le nom de file vient d'une analogie avec une file d'attente à un guichet, dans laquelle le premier arrivé sera la premier servi. Les usagers arrivent en queue de file, et sortent de la file à sa tête.

Une file peut être implémentée par une liste chaînée, ou par un tableau avec une gestion circulaire. Comme dans le cas des piles, la gestion par tableaux présente l'inconvénient que la file a une capacité limitée, contrairement à la gestion par listes chaînées.

Comme dans le cas des piles, on gère les files à l'aide de primitives. Les primitives de gestion des files sont les suivantes :

- `Initialiser` : cette fonction crée une file vide.
- `EstVide` : renvoie 1 si la file est vide, 0 sinon.
- `EstPleine` : renvoie 1 si la file est pleine, 0 sinon.
- `AccederTete` : cette fonction permet l'accès à l'information contenue dans la tête de file.
- `Enfiler` : cette fonction permet d'ajouter un élément en queue de file. La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.
- `Defiler` : cette fonction supprime la tête de file. L'élément supprimé est retourné par la fonction `Defiler` pour pouvoir être utilisé.
- `Vider` : cette fonction vide la file.
- `Detruire` : cette fonction permet de détruire la file.

Comme dans le cas des piles, lorsqu'on utilise une file, on ne se préoccupe pas de la manière dont elle a été implémentée, mais on utilise uniquement les primitives qui sont toujours les mêmes.

21.2 GESTION NAÏVE PAR TABLEAUX

Les primitives dans une mauvaise (mais facile) gestion des files par tableaux sont les suivantes (voir la figure 21.1) :

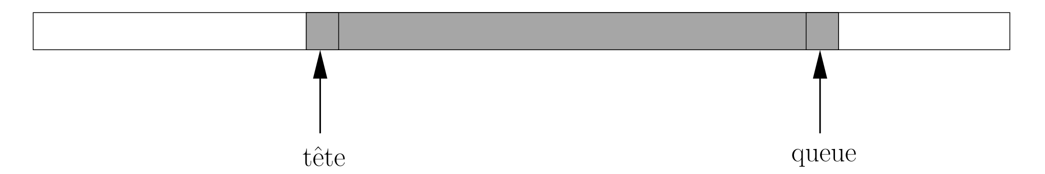


Figure 21.1– Gestion naïve d'une file par tableaux

```
typedef float TypeDonnee;

typedef struct
{
  int nb_elem_max;  /* nombre d'éléments du tableau */
  int indice_tete, indice_queue; /* indice de tête et queue */
  TypeDonnee *tab; /* tableau des éléments */
}File;
```

```
File Initialiser(int nb_max)
{
  File filevide;
  filevide.indice_tete=0;   /* la pile est vide */
  filevide.indice_queue=-1;
  filevide.nb_elem_max = nb_max;  /* capacité nb_max */
  /* allocation des éléments : */
  filevide.tab = (TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
  return filevide;
}
```

```
int EstVide(File F)
{
  /* renvoie 1 si l'indice de queue est plus petit */
  return (F.indice_tete == F.indice_queue+1) ? 1 : 0;
}
```

```
int EstPleine(File F)
{
  /* la file est pleine quand la queue est au bout du tableau */
  return (F.indice_queue == F.nb_elem_max-1) ? 1 : 0;
}
```

```
int AccederTete(File F, TypeDonnee *pelem)
{
  if (EstVide(F))
    return 1;   /* on retourne un code d'erreur */
  *pelem = F.tab[F.indice_tete]; /* on renvoie l'élément */
  return 0;
}
```

```
void Vider(File *pF)
{
  pF->indice_tete = 0;  /* réinitialisation des indices */
  pF->indice_queue = -1;
}
```

```
void Detruire(File *pF)
{
  if (pF->nb_elem_max != 0)
    free(pF->tab);  /* libération de mémoire */
  pF->nb_elem_max = 0; /* file de taille 0 */
}
```

```
char Enfiler(File *pF, TypeDonnee elem)
{
  if (pF->indice_queue >= pF->nb_elem_max-1)
    return 1;   /* erreur : file pleine */
  pF->indice_queue++; /* on insère un élément en queue */
  pF->tab[pF->indice_queue] = elem; /* nouvel élément */
  return 0;
}
```

```
char Defiler(File *pF, TypeDonnee *pelem)
{
  if (pF->indice_tete == pF->indice_queue-1)
    return 1;   /* erreur : file vide */
  *pelem = pF->tab[pF->indice_tete]; /* on renvoie la tête */
  pF->indice_tete++; /* supprime l'élément de tête */
  return 0;
}
```

Le problème dans cette gestion des files par tableaux est qu'à mesure qu'on insère et qu'on supprime des éléments, les indice de tête et de queue ne font qu'augmenter. L'utilisation d'une telle file est donc limitée dans le temps, et `nb_elem_max` est le nombre total d'éléments qui pourront jamais être insérés dans la file.

21.3 Gestion circulaire par tableaux

21.3.1 Enfiler et défiler

Dans une gestion circulaire des files, lorsque le bout du tableau est atteint, on réutilise la mémoire du début du tableau (voir la figure 21.2). Pour cela, on utilise des modulo sur les indices du tableau.

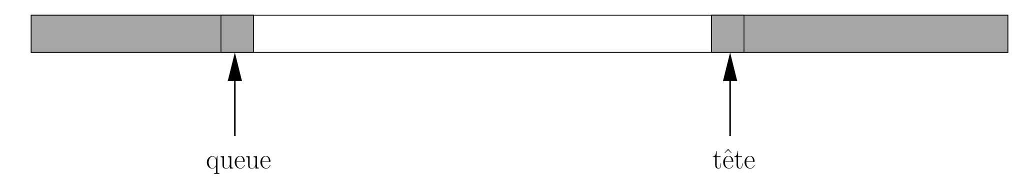


Figure 21.2 – Gestion circulaire d'une file par tableaux

```
char Enfiler(File *pF, TypeDonnee elem)
{
   if (EstPleine(*pF))
     return 1; /* on ne peut rien ajouter à une file pleine */
   pF->indice_queue++; /* insertion en queue de file */
   if (pF->indice_queue == pF->nb_elem_max) /* si au bout */
     pF->indice_queue = 0; /* on réutilise le début */
   pF->tab[pF->indice_queue] = elem; /* ajout de l'élément */
   return 0;
}
```

```
char Defiler(File *pF, TypeDonnee *pelem)
{
  if (EstVide(*pF))
    return 1;  /* on ne peut pas supprimer d'élément */
  *pelem = pF->tab[pF->indice_tete++]; /* renvoie l'élément */
  if (pF->indice_tete == pF->nb_elem_max) /* si on est au bout */
    pF->indice_tete = 0; /* on passe au début du tableau */
  return 0;
}
```

21.3.2 Autres primitives

Les primitives suivantes fonctionnent dans la gestion circulaire.

```
File Initialiser(int nb_max)
{
  File filevide;
```

```
  filevide.indice_tete=1;   /* la pile est vide */
  filevide.indice_queue=0;
  filevide.nb_elem_max = nb_max;  /* capacité nb_max */
  /* allocation des éléments : */
  filevide.tab = (TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
  return filevide;
}
```

```
int EstVide(File F)
{
  /* on utilise un modulo si l'indice dépasse F.nb_elem_max */
  if (F.indice_tete == (F.indice_queue+1)%F.nb_elem_max)
   return 1;
  else
    return 0;
}
```

La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(File F)
{
  /* la file est pleine si on ne peut pas rajouter d'élément */
  /* sans obtenir la condition de file vide */
  if (F.indice_tete == (F.indice_queue+2)%F.nb_elem_max)
    retrun 1;
  else
    return 0;
}
```

```
int AccederTete(File F, TypeDonnee *pelem)
{
  if (EstVide(F))
    return 1;  /* on retourne un code d'erreur */
  *pelem = F.tab[F.indice_tete]; /* on renvoie l'élément */
  return 0;
}
```

```
void Vider(File *pF)
{
  pF->indice_tete = 1; /* réinitialisation des indices */
  pF->indice_queue = 0;
}
```

```
void Detruire(File *pF)
{
  if (pF->nb_elem_max != 0)
    free(pF->tab);   /* libération de mémoire */
  pF->nb_elem_max = 0; /* file de taille 0 */
}
```

Dans la gestion des files par tableaux, le nombre d'éléments qui peuvent être présents simultanément dans la file est limité par `nb_elem_max`. Voyons maintenant la gestion par listes chaînées qui permet d'allouer de la mémoire à mesure des besoins.

21.4 Gestion par listes chaînées

21.4.1 Structures de données

Pour la gestion par listes chaînées, on introduit un pointeur sur la tête de liste et un pointeur sur la queue de liste (voir figure 21.3). Ceci permet de faire les insertions en queue de liste sans avoir à parcourir la liste pour trouver l'adresse de la dernière cellule (voir l'insertion en queue de liste classique au chapitre 19).

```
typedef float TypeDonnee;

typedef struct Cell  /* déclaration de la liste chaînée */
{
  TypeDonnee donnee;
  struct Cell *suivant;
}TypeCellule;

typedef struct
{
  TypeCellule *tete, *queue; /* pointeurs sur la première */
                             /* et dernière cellule */
}File;
```

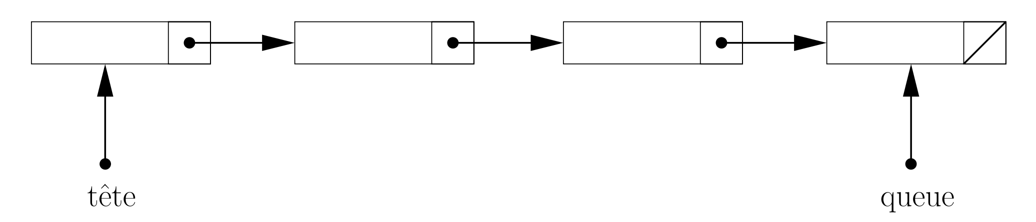


Figure 21.3 – Gestion d'une file par listes chaînées

21.4.2 Primitives

```
File Initialiser()
{
   File filevide;
   filevide.tete=NULL;  /* liste vide : NULL */
}
```

```
int EstVide(File F)
{
  return (F.tete == NULL) ? 1 : 0;
}
```

```
int EstPleine(File F)
{
  return 0; /* une liste chaînée n'est jamais pleine */
}
```

```
int AccederTete(File F, TypeDonnee *pelem)
{
  if (EstVide(F))
    return 1;   /* on retourne un code d'erreur */
  *pelem = F.tete->donnee;  /* on renvoie la donnée de tête */
  return 0;
}
```

```
void Detruire(File *pF)
{
  TypeCellule *p, *q;
  p=pF->tete; /* initialisation pour parcours de liste */
  while (p!= NULL)
    {
      q = p;  /* mémorisation de l'adresse */
      p = p->suivant; /* passage au suivant avant destruction */
      free(q); /* destruction de la cellule */
    }
  pF->tete = NULL; /* on met la liste à vide */
}
```

```
void Vider(File *pF)
{
  Detruire(pF);  /* destruction de la liste */
  pF->tete = NULL; /* liste vide */
}
```

```
void Enfiler(File *pF, TypeDonnee elem)
{
   TypeCellule *q;
   q = (TypeCellule*)malloc(sizeof(TypeCellule)); /* allocation */
   q->donnee = elem;
   q->suivant = NULL;  /* suivant de la dernière cellule NULL */
   if (pF->tete == NULL)  /* si file vide */
     {
       pF->queue = pF->tete = q;
     }
   else
     {
       pF->queue->suivant = q;  /* insertion en queue de file */
       pF->queue = q;
     }
}
```

```
char Defiler(File *pF, TypeDonnee *pelem)
{
  TypeCellule *p;
  if (EstVide(*pF))
    return 1;        /* retour d'un code d'erreur */
  *pelem = pF->tete->donnee;  /* on renvoie l'élément */
  p = pF->tete;  /* mémorisation de la tête de liste */
  pF->tete = pF->tete->suivant; /* passage au suivant */
  free(p);  /* destruction de l'ancienne tête de liste */
  return 0;
}
```

Exercices

21.1 (∗∗) Dans une gare, un guichet est ouvert. Les clients arrivent à des dates aléatoires et rentrent dans une queue. L'intervalle entre l'arrivée de deux clients successifs est un nombre aléatoire entre 0 et `INTERVALLE_MAX` (les dates sont des entiers indiquant des secondes). Lorsque le guichetier a fini de traiter un client, il appelle le client suivant dont le traitement va avoir une durée aléatoire entre 0 et `DUREE_TRAITEMENT_MAX`.

a) Définir les structures de données pour l'algorithme de simulation.

b) Écrire une fonction *CreerListeClients*, qui crée une file de clients, le nombre de clients étant saisi au clavier. Cette fonction initialise aussi la date d'arrivée et la durée d'attente de chacun des clients. On supposera que le premier client est arrivé à 8h.

c) Écrire une fonction d'affichage qui affiche le numéro de chacun des clients, sa date d'arrivée et sa date de fin de traitement en format (h min sec).

Corrigés

21.1

```
#define INTERVALLE_MAX        180
#define DUREE_TRAITEMENT_MAX        600
```

a)

```
  typedef struct
{
  int numero;
  int datearrivee;
  int dureetraitement;
} TypeClient;
typedef struct Cell
{
  TypeClient donnee;
  struct Cell *suivant;
} TypeCellule;
typedef struct
{
  TypeCellule *tete, *queue;
} File;
```

b)

```
void Enfiler(File * pF, TypeClient pelem)
{
  TypeCellule *q;
  q = (TypeCellule *) malloc(sizeof(TypeCellule));
  q->donnee.numero = pelem.numero;
  q->donnee.datearrivee = pelem.datearrivee;
  q->donnee.dureetraitement = pelem.dureetraitement;
  q->suivant = NULL;
  if (pF->tete == NULL)
    pF->queue = pF->tete = q;
  else
    {
```

```
      pF->queue->suivant = q;
      pF->queue = q;
    }
}
void CreerListeClients(File * pF)
{
  int i, nbclient;
  int dernieredate = 8 * 60 * 60;        /*le 1er client arrive à 8h */
  TypeClient client;
  printf("Entrer le nombre de clients\n");
  scanf("%d", &nbclient);
  for (i = 0; i < nbclient; i++)
    {
      client.numero = i + 1;
      dernieredate += rand() % INTERVALLE_MAX;
      client.datearrivee = dernieredate;
      client.dureetraitement = rand() % DUREE_TRAITEMENT_MAX;
      Enfiler(pF, client);
    }
}
```

c)

```
char Affichage(File * pF)
{
  int fintraitement = 0, tmp;
  TypeCellule *q;
  q = pF->tete;
  if (q == NULL)
    return 1;
  while (q != NULL)
    {
      if (q->donnee.datearrivee > fintraitement)
        fintraitement = q->donnee.datearrivee;
      fintraitement += q->donnee.dureetraitement;
      tmp = q->donnee.datearrivee;
      /* Affichage convertit de "sec" en format "h min sec" */
      printf("Client %d: est arrivé à %d h %d mn %d sec",
             q->donnee.numero,
             tmp / 3600, (tmp % 3600) / 60, (tmp % 3600) % 60);
      printf(" et a fini à %d h %d mn %d sec\n", fintraitement / 3600,
             (fintraitement % 3600) / 60, (fintraitement % 3600) % 60);
      q = q->suivant;
    }
  return 0;
}
```

Récursivité 22

22.1 Qu'est-ce que la récursivité ?

Une fonction *C* est dite *récursive* si elle s'appelle elle-même.

Exemple

La fonction suivante calcule la factorielle $n!$ d'un nombre n. On se base sur la relation de récurrence $n! = (n-1)! * n$

```
int FactorielleRecursive(int n)
{
   if (n <= 0)
      return 1;
   else
      return n*FactorielleRecursive(n-1);   /* appel récursif */
}
```

L'appel d'une fonction à l'intérieur d'elle-même est nommé *appel récursif*. Un appel récursif doit obligatoirement être dans une instruction conditionnelle (sinon la récursivité est sans fin).

Remarque

On aurait aussi pu implémenter la fonction factorielle de manière itérative :

```
int FactorielleIterative(int n)
{
   int i, resultat=1;
   for (i=1 ; i<=n ; i++)
      resultat *= i;
   return resultat;
}
```

Lorsque c'est possible, on préfèrera la version itérative, qui consomme moins de mémoire, car une fonction récursive utilise une pile d'appels (voir section 22.3).

22.2 Comment programmer une fonction récursive ?

22.2.1 Résolution récursive d'un problème

Pour créer une fonction récursive, il faut :

1. décomposer un problème en un ou plusieurs sous-problèmes du même type. On résoud les sous-problèmes par des appels récursifs.

2. Les sous-problèmes doivent être de taille plus petite que le problème initial.
3. Enfin, la décomposition doit en fin de compte conduire à un cas élémentaire, qui, lui, n'est pas décomposé en sous-problème. (condition d'arrêt).

> ***Exemple***
>
> Problème : calculer $n!$.
> Sous-problème : calculer $(n-1)!$. On multipliera le résultat de la résolution du sous-problème par n.
> Le calcul de $(n-1)!$ passe par le calcul de $(n-2)!$, etc...
> À la fin, le problème devient le calcul de 0!. C'est un problème élémentaire : le résultat vaut 1.

22.2.2 Structure d'une fonction récursive

```
FONCTION FonctionRécursive(type1 p1, type2 p2,..., typek pk)
                           : typeretour
début
   si condition  faire           /* condition d'arrêt */
      retourner calcul;    /* cas élémentaire */
   sinon faire
      FonctionRécursive(...);  /* appel récursif */
      ...
      FonctionRécursive(...);  /* appel récursif */
      retourner quelque-chose;
   fin faire
fin
```

22.3 Pile d'appels

Un appel d'une fonction récursive ne se termine pas avant que tous les sous-problèmes soient résolus. Pendant tout ce temps, les paramètres et variable locales de cet appel sont stockés dans une pile. Cette pile est appelée *pile d'appels* (en anglais *stack*). Le programmeur n'a pas à se préoccuper de la pile d'appels car celle-ci est gérée automatiquement par le système.

> ***Exemple***
>
> Pour calculer $n!$ on fait un appel de `FactorielleRecursive(n)`. Dans cet appel, on calcule $(n-1)!$ récursivement. Pendant tout le calcul de $(n-1)!$, il faut se rappeler de la valeur de n pour multiplier $(n-1)!$ par n une fois calculé $(n-1)!$. La valeur de n est automatiquement stockée dans la pile d'appels.
>
> De même, pour calculer $(n-1)!$, on va calculer $(n-2)!$. Pendant le calcul de $(n-2)!$, la valeur $n-1$ est stockée dans la pile. Les valeurs sont dépilées à mesure que les appels récursifs se terminent.

Compléments

√ La gestion des variables locales dans une pile d'appel n'est pas limitée aux fonctions récursive. Lors de l'exécution d'un programme, la fonction `main` appelle d'autres fonctions, qui appellent à leur tour des fonctions, etc. À un instant donné, les fonctions en cours d'exéctution s'appellent les unes les autres. Ces fonctions, et leurs variables locales, sont stockées dans une pile, appelée pile d'appel (en anglais *call stack*). La fonction `main` est tout en bas de la pile et la fonction contenant l'instruction en train de s'exécuter est au sommet de la pile. Lorsqu'une fonction se termine, ses variables sont dépilées (les variables locales sont détruites). Lorsque que le programme rentre dans une fonction, ses variables locales sont créées au sommet de la pile. La gestion *LIFO* sous forme de pile est parfaitement adaptée à la gestion mémoire des variables locales des fonctions.

Exercices

22.1 (∗) Écrire une fonction récursive pour calculer la somme :

$$u_n = 1 + 2^4 + 3^4 + \cdots + n^4$$

Observer les variables mémorisées pour le calcul de u_6.

22.2 (∗) La suite de Fibonacci est définie par récurrence par :

$$u_0 = 0\ ;\ u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2} \text{ pour } n \geq 2$$

a) Donner un algorithme récursif pour calculer u_n. Combien y-a-t'il d'appels récursifs pour le calcul de u_6 ?

b) Combien y a-t-il d'appels récursifs pour calculer u_n ? Quelle est la complexité de l'algorithme récursif calculant u_n ?

c) Donner un algorithme itératif pour calculer u_n et donner la complexité de cet algorithme. Que peut-on en conclure ?

22.3 (∗)

a) Donner un algorithme récursif pour afficher une liste chaînée à l'envers. Observer les états successifs de la mémoire lors de l'affichage d'une liste à 5 éléments.

b) Donner aussi un algorithme itératif.

22.4 (∗) Les coefficients binomiaux C_n^k pour $k \leq n$ sont donnés par la formule de Pascal :

$$C_n^0 = 1 \; ; \; C_n^n = 1$$

$$C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

a) Proposer un algorithme récursif pour calculer les coefficients binomiaux. Observer les états successifs de la mémoire pour le calcul de C_5^3.

b) Proposer un algorithme itératif pour calculer les coefficients binomiaux.

22.5 (∗∗) **(tours de Hanoï)** Une tour de Hanoï est une tableau `T` de n entiers tel que `T[0]` < `T[1]` < · · · < `T[n − 1]`. Le jeu des tours de Hanoï consiste à déplacer une tour de Hanoï `T1` dans une tour de Hanoï `T2` (qui est initialement vide) en utilisant une tour de Hanoï `T3` (qui est initialement vide). Les règles du jeu sont les suivantes :

1. À chaque étape seul 1 élément peut être déplacé du sommet (dernier élément) d'une tour vers le sommet d'une autre tour.
2. À chaque étape, `T1`, `T2` et `T3` sont des tours de Hanoï.

Proposer un algorithme pour résoudre le problème des tours de Hanoï.

22.6 (∗∗) Dans une expression arithmétique au format préfixé, on trouve d'abord l'opérateur +, −, ∗, /, puis les opérandes. Par exemple, l'expression parenthésée (10 + 2) sera écrite :

+ 10 2

Les opérandes peuvent être eux-mêmes des expressions. Par exemple, l'expression parenthésée ((10 + 2)/3) sera écrite :

/ + 10 2 3

a) Donner un algorithme récursif pour évaluer le résultat d'une expression préfixée.

b) Donner un algorithme permettant de traduire une expression préfixée en expression parenthésée.

c) Donner un algorithme permettant de traduire une expression parenthésée en expression préfixée. On supposera que chaque opération $x + y$, $x - y$, $x * y$ ou x/y de l'expression est écrite entre parenthèses.

Corrigés

22.1

```
int Somme(int n)
{
  if (n <= 0)
    return 0;
  else
    return (n * n * n * n + Somme(n - 1));
}
```

22.2

```
int Fibonacci(int n)
{
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return (Fibonacci(n - 1) + Fibonacci(n - 2));
}
```

Pour calculer u_6, il en faut 25 appels récursifs

22.3

a)

```
void AffichageRecursif(TypeCellule * L)
{
  TypeCellule *p;
  p = L;
  if (p != NULL)
    {
      printf("%d\t", p->donnee);
      p = p->suivant;
      return AffichageRecursif(p);
    }
  puts("");
}
```

b)

```
void AffichageIteratif(TypeCellule * L)
{
  TypeCellule *p;
  p = L;
  while (p != NULL)
    {
      printf("%d\t", p->donnee);
      p = p->suivant;
    }
  puts("");
}
```

22.4

a)

```
int Coeffbinom(int k, int n)
{
  if (k == 0 || k == n)
    return 1;
  else
    return Coeffbinom(k - 1, n - 1) + Coeffbinom(k, n - 1);
}
```

b)

```
/*Cette fonction calcul tous les coefficients
 binomiaux les mettant dans tab*/
void Coeffbinom(int k, int n, int **tab)
{
  int i, j;
  for (i = 0; i <= n; i++)
    {
      tab[i][0] = 1;
      tab[i][i] = 1;
    }
  for (i = 2; i <= n; i++)
    for (j = 1; j < i; j++)
      tab[i][j] = tab[i - 1][j - 1] + tab[i - 1][j];
}
```

22.5

```
void deplacer(int n, int T1, int T2, int T3)
{
  if (n == 1)
```

```
    printf("De la tour %d à la tour %d\n", T1, T2);
  else
    {
      deplacer(n - 1, T1, T3, T2);
      deplacer(1, T1, T2, T3);
      deplacer(n - 1, T3, T2, T1);
    }
}
```

22.6

a)

```
typedef char TypeDonnee;
typedef struct Cell
{
  TypeDonnee donnee[20];
  struct Cell *suivant;
} TypeCellule;
typedef TypeCellule *Pile;
/* Créer une pile vide */
Pile initialiser()
{
  return NULL;
}
char EstVide(Pile P)
{
  return (P == NULL) ? 1 : 0;
}
char EstPleine(Pile P)
{
  return 0;
}
void Empiler(Pile * pP, TypeDonnee * elem)
{
  Pile q;
  q = (TypeCellule *) malloc(sizeof(TypeCellule));
  strcpy(q->donnee, elem);
  q->suivant = *pP;
  *pP = q;
}
 char Depiler(Pile * pP, TypeDonnee * pelem)
{
  Pile q;
  if (EstVide(*pP))
```

```
    return 1;
  strcpy(pelem, (*pP)->donnee);
  q = *pP;
  *pP = (*pP)->suivant;
  free(q);
  return 0;
}
void Afficherpile(Pile P)
{
  Pile q;
  q = P;
  while (q != NULL)
    {
      printf("%s\t", q->donnee);
      q = q->suivant;
    }
  puts("");
}
void EvalPrefixe(Pile * Original, Pile * Cumul)
{
  Pile q;
  TypeDonnee a[20], b[20], c[20];
  q = *Original;
  if (q != NULL)
    {
      if (strcmp(q->donnee, "+") == 0)
        {
          Depiler(Cumul, a);
          Depiler(Cumul, b);
          sprintf(c, "%d", (atoi(a) + atoi(b)));
          Empiler(Cumul, c);
        }
      else if (strcmp(q->donnee, "*") == 0)
        {
          Depiler(Cumul, a);
          Depiler(Cumul, b);
          sprintf(c, "%d", (atoi(a) * atoi(b)));
          Empiler(Cumul, c);
        }
      else if (strcmp(q->donnee, "-") == 0)
        {
          Depiler(Cumul, a);
          Depiler(Cumul, b);
          sprintf(c, "%d", (atoi(a) - atoi(b)));
```

```
            Empiler(Cumul, c);
          }
        else if (strcmp(q->donnee, "/") == 0)
          {
            Depiler(Cumul, a);
            Depiler(Cumul, b);
            sprintf(c, "%d", (atoi(a) / atoi(b)));
            Empiler(Cumul, c);
          }
        else
          Empiler(Cumul, q->donnee);
        q = q->suivant;
        return EvalPrefixe(&q, Cumul);
      }
}
```

b)

```
void ExprParenthese(Pile * Original, Pile * Cumul)
{
  Pile q;
  char a[20], b[20], c[20];
  q = *Original;
  if (q != NULL)
    {
      if (strcmp(q->donnee, "+") == 0 || strcmp(q->donnee, "-") == 0
         || strcmp(q->donnee, "*") == 0 || strcmp(q->donnee, "/") == 0)
        {
          strcpy(c, "(");
          Depiler(Cumul, a);
          Depiler(Cumul, b);
          strcat(c, a);
          strcat(c, q->donnee);
          strcat(c, b);
          strcat(c, ")");
          Empiler(Cumul, c);
        }
      else
        Empiler(Cumul, q->donnee);
      q = q->suivant;
      return ExprParenthese(&q, Cumul);
    }
}
```

c)

```
void ParentheseEnPrefixe(Pile * Original, Pile * Cumul, Pile * final)
{
  Pile q;
  char a[20];
  q = *Original;
  if (q != NULL)
    {
      if (strcmp(q->donnee, "+") == 0 || strcmp(q->donnee, "-") == 0
         || strcmp(q->donnee, "*") == 0 || strcmp(q->donnee, "/") == 0)
        {
          Empiler(Cumul, q->donnee);
        }
      else if (strcmp(q->donnee, ")") == 0)
        {
        }
      else if (strcmp(q->donnee, "(") == 0)
        {
          Depiler(Cumul, a);
          Empiler(final, a);
        }
      else
        {
          Empiler(final, q->donnee);
        }
      q = q->suivant;
      return ParentheseEnPrefixe(&q, Cumul, final);
    }
}
```

ARBRES BINAIRES 23

23.1 QU'EST-CE QU'UN ARBRE BINAIRE ?

Un *arbre binaire* fini est un ensemble fini de cellules, chaque cellule étant liée à 0, 1 ou 2 autres cellules appelées *cellules filles*. Dans un arbre, toutes les cellules sauf une ont exactement une cellule mère. Si l'arbre est non vide, une seule cellule n'a pas de cellule mère, et celle-ci est appelée *racine* de l'arbre (voir figure 23.1). Enfin, un arbre est connexe, c'est-à-dire que toute cellule est descendante de la racine.

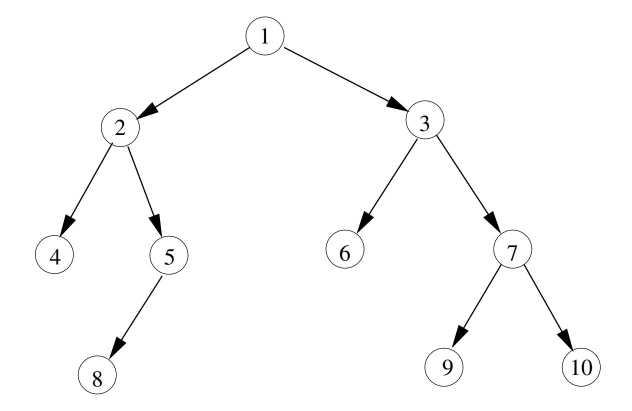


Figure 23.1 – **Un arbre binaire**

Dans la terminologie des arbres, les cellules sont appelées des *nœuds* ou des *sommets*. Un nœud n'ayant pas de fils s'appelle une *feuille*.

En langage *C*, on représente un nœud d'un arbre comme une structure, chaque nœud contenant deux pointeurs vers les éventuels nœuds fils (voir figure 23.2).

En d'autres termes, un arbre binaire est une structure analogue à une liste chaînée, sauf que chaque cellule possède deux suivants. Par convention, on convient d'appeler *fils gauche* et *fils droit* les nœuds fils d'un nœud. Les fils gauche et fils droit d'un nœud peuvent ne pas exister (pointeur égal à NULL). L'accès à l'arbre est donné par pointeur qui contient l'adresse de la racine.

La structure de données précise pour représenter un arbre binaire peut être définie comme suit :

```
typedef int TypeDonnee;

typedef struct cell
```

```
{
   TypeDonnee donnee;
   struct cell *fg, *fd;  /* fils gauche et droit */
}TypeNoeud;
```

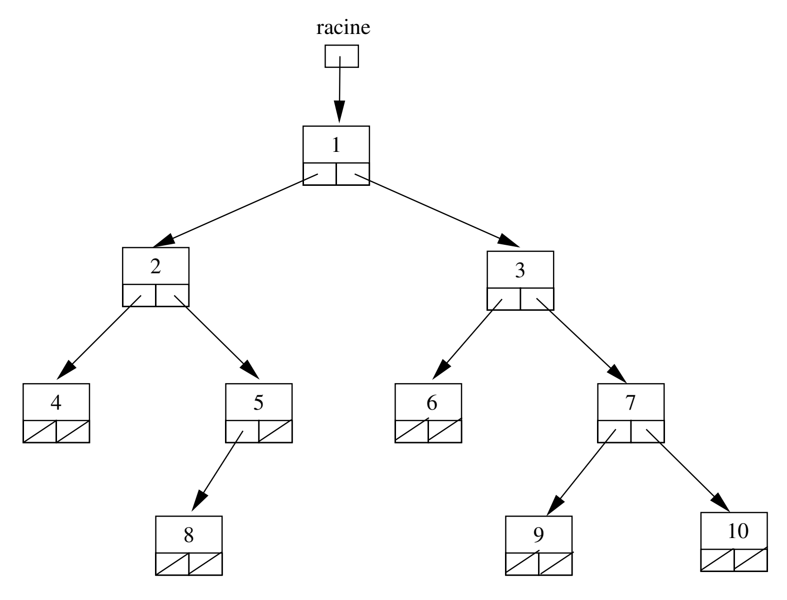


Figure 23.2 – Un arbre binaire représenté par des structures en *C*

On appelle sous-arbre d'un nœud *N* l'arbre dont la racine est *N* et dont les nœuds sont les descendants de *N*. Chaque nœud *N* possède aussi un sous-arbre gauche, qui est l'ensemble des descendants de son fils gauche, et un sous-arbre droit, qui est l'ensemble des descendants de son fil droit. Les sous-arbres gauche et droit de l'arbre peuvent être vides si les fils gauche ou droit du nœud n'existent pas (pointeurs `fg` ou `fd` égaux à `NULL`).

23.2 Parcours d'arbres binaires

Pour afficher un arbre ou rechercher un élément dans un arbre, on doit parcourir l'arbre, c'est à dire examiner tous ses nœuds. Le moyen le plus simple de parcourir un arbre est de faire une fonction récursive. Les parcours portent des noms différents selon l'ordre dans lequel on examine les nœuds.

23.2.1 Parcours préfixé

Dans le parcours préfixé, on traite d'abord la racine de l'arbre, puis on parcourt récursivement le sous-arbre gauche et le sous-arbre droit de la racine.

Exemple

Pour l'arbre de la figure 23.1, le parcours préfixé consiste à :

1. Traiter la racine 1 ;
2. Parcourir le sous-arbre gauche :
   a) Traiter la racine 2
   b) Parcourir le sous-arbre gauche
      i) Traiter la racine 4 ;
      ii) Le sous-arbre gauche est vide ;
      iii) Le sous-arbre droit est vide ;
   c) Parcourir le sous-arbre droit :
      i) Traiter la racine 5 ;
      ii) Parcourir le sous-arbre gauche :
         α) Traiter la racine 8 ;
         β) Les sous-arbres gauche et droit sont vides ;
      iii) Le sous-arbre droit est vide ;
3. Parcourir le sous-arbre droit :
   a) Traiter la racine 3 ;
   b) Parcourir le sous-arbre gauche :
      i) Traiter la racine 6 ;
      ii) Les sous-arbres gauche et droit sont vides ;
   c) Parcourir le sous-arbre droit :
      i) Traiter la racine 7 ;
      ii) Parcourir le sous-arbre gauche :
         α) Traiter la racine 9 ;
         β) Les sous-arbres gauche et droit sont vides ;
      iii) Parcourir le sous-arbre droit :
         α) Traiter la racine 10 ;
         β) Les sous-arbres gauche et droit sont vides ;

Les nœuds sont donc parcourus dans l'ordre : $1, 2, 4, 5, 8, 3, 6, 7, 9, 10$

L'algorithme de parcours préfixé est le suivant :

```
void ParcoursPrefixe(TypeNoeud *racine)
{
   if (racine != NULL)
```

```
        {
            Traiter(racine);
            ParcoursPrefixe(racine->fg);
            ParcoursPrefixe(racine->fd);
        }
}
```

Par exemple, pour un affichage de données de type `int` contenues dans l'arbre, on fera la fonction `Traiter` suivante :

```
void Traiter(TypeNoeud *noeud)
{
    printf("%d, ", noeud->donnee);
}
```

23.2.2 Parcours postfixé

Dans le parcours postfixé, on effectue les choses dans l'ordre suivant :

1. Parcourir le sous-arbre gauche ;
2. Parcourir le sous-arbre droit ;
3. Traiter la racine.

Ainsi, l'algorithme de parcours postfixé envisage les nœuds de l'arbre de la figure 23.1 dans l'ordre suivant : 4, 8, 5, 2, 6, 9, 10, 7, 3, 1.
L'algorithme en *C* est le suivant :

```
void ParcoursPostfixe(TypeNoeud *racine)
{
    if (racine != NULL)
        {
            ParcoursPostfixe(racine->fg);
            ParcoursPostfixe(racine->fd);
            Traiter(racine);
        }
}
```

23.2.3 Parcours infixé

Dans le parcours infixé (ou parcours symétrique), on effectue les choses dans l'ordre suivant :

1. Parcourir le sous-arbre gauche ;
2. Traiter la racine.
3. Parcourir le sous-arbre droit ;

Ainsi, l'algorithme de parcours infixé envisage les nœuds de l'arbre de la figure 23.1 dans l'ordre suivant : 4, 2, 8, 5, 1, 6, 3, 9, 7, 10.
L'algorithme en *C* est le suivant :

```
void ParcoursInfixe(TypeNoeud *racine)
{
   if (racine != NULL)
      {
         ParcoursInfixe(racine->fg);
         Traiter(racine);
         ParcoursInfixe(racine->fd);
      }
}
```

23.3 Libération de mémoire

Pour libérer la mémoire d'un arbre, on doit parcourir l'arbre et libérer chaque cellule par un appel à `free`. Pour plus de sûreté, la fonction de libération de mémoire mettra la racine à `NULL`, pour avoir ensuite un arbre vide correct. Ainsi, l'arbre libéré ne provoquera pas d'erreur mémoire en cas d'utilisation (par exemple dans un parcours). Pour cela, il faut passer la racine par adresse.

```
void Liberer(TypeNoeud **p_racine)
{
   TypeNoeux *racine = *p_racine;
   if (racine != NULL)
      {
         Liberer(&racine->fg);
         Liberer(&racine->fd);
         free(racine);
      }
   *p_racine = NULL.
}
```


Exercices

23.1 (∗) Écrire une fonction qui recherche un nombre n dans un arbre d'entiers. La fonction doit renvoyer 1 si n est présent, 0 sinon.

23.2 (∗) Écrire une fonction qui calcule la somme des valeurs des nœuds dans un arbre de `float`.

23.3 (∗) Écrire une fonction calculant le maximum des valeurs des nœuds d'un arbre d'entiers.

23.4 (∗∗) On représente une expression arithmétique par un arbre dont les nœuds sont des opérateurs ou des nombres et dont les feuilles sont des nombres.

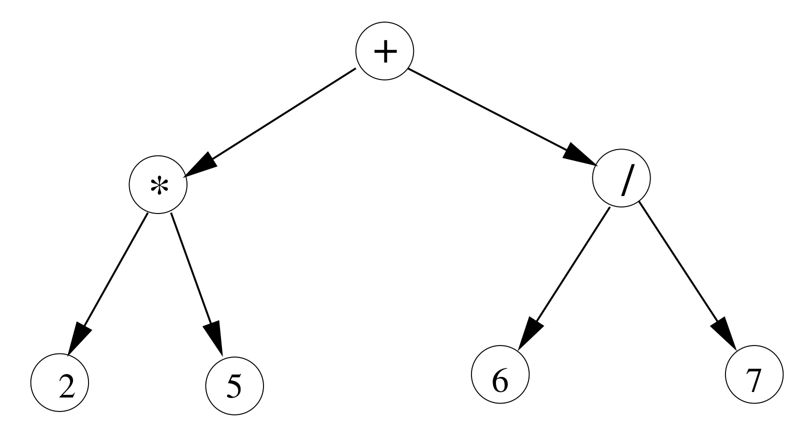


Figure 23.3 – L'arbre associé à l'expression $(2 * 5) + (6/7)$.

a) Écrire les structures de données permettant de représenter une expression arithmétique sous forme d'arbre.

b) Écrire une fonction d'évaluation d'une expression arithmétique sous forme d'arbre.

c) Écrire une fonction qui génère une chaîne de caractères contenant l'expression préfixée correspondant à un arbre.

d) Écrire une fonction de création d'un arbre représentant une expression arithmétique donnée au format préfixé.

e) Écrire une fonction d'écriture d'une expression arithmétique sous forme parenthésée à partir d'un arbre.

f) Écrire une fonction de création d'un arbre représentant une expression arithmétique donnée au format parenthésé.

23.5 (∗ ∗ ∗) **(Arbres lexicographiques)** On représente un dictionnaire sous forme d'arbre binaire. Les premières lettres possibles pour le mot sont obtenues en suivant le branche de droite à partir de la racine (par exemple, sur la figure 23.4, les premières lettres possibles sont *a*, *b* ou *c*).

Figure 23.4 – **Dictionnaire contenant les mots : art, as, bas, beau, car, cas**

Pour accéder aux deuxièmes lettres des mots, on passe au fils gauche de la première lettre. Toutes les deuxièmes lettres possibles sont alors obtenues en prenant la branche de droite (Par exemple, sur la figure 23.4, avec la première lettre *b*, les deuxième lettres possibles sont *a* et *e*). Pour avoir la troisième lettre, on passe au fils gauche de la deuxième lettre, et ainsi de suite jusqu'à trouver un `\0`.

a) Proposer une structure de données pour représenter un dictionnaire.

b) Écrire une fonction récursive qui vérifie si un mot est dans le dictionnaire.

c) Écrire une fonction itérative qui vérifie si un mot est dans le dictionnaire.

d) Écrire une fonction qui vérifie l'othographe d'un fichier texte, et affiche les mots qui ne sont pas dans le dictionnaire. On suppose que les mots sont séparés par des espaces ou des retours à la ligne.

e) Écrire une fonction qui rajoute un mot *m* dans le dictionnaire. On cherchera la dernière lettre du plus long sous-mot (qui soit le début du mot *m*) qui est présent dans le doctionnaire, et on cherchera en même temps l'adresse du noeud de l'arbre correspondant. On ajoutera ensuite des descendants à ce noeud.

f) Écrire une fonction qui sauvegarde un dictionnaire sous forme de liste de mots dans un fichier texte.

g) Écrire une fonction qui charge en mémoire un dictionnaire donné sous forme de liste de mots dans un fichier texte.

h) Que faudrait-il faire pour que les mots du dictionnaire sauvegardé soit rangés dans l'ordre alphabétique ?

Corrigés

23.1

```
void Recherchen(TypeNoeud * racine, int n, int *retour)
{
  if (racine != NULL)
    {
      if (racine->donnee == n)
        *retour = 1;
      Recherchen(racine->fg, n, retour);
      Recherchen(racine->fd, n, retour);
    }
}
```

23.2

```
typedef float TypeDonnee;
typedef struct Cell
{
  TypeDonnee donnee;
  struct Cell *fg, *fd;
} TypeNoeud;
void CalculSomme(TypeNoeud * racine, float *somme)
{
  if (racine != NULL)
    {
      *somme = *somme + racine->donnee;
      CalculSomme(racine->fg, somme);
      CalculSomme(racine->fd, somme);
    }
}
```

23.3

```
void CalculMax(TypeNoeud * racine, int *maximum)
{
  if (racine != NULL)
    {
```

```
      *maximum = racine->donnee > *maximum ? racine->donnee : *maximum;
      CalculMax(racine->fg, maximum);
      CalculMax(racine->fd, maximum);
    }
}
```

23.4

a)

```
typedef char TypeDonnee;
typedef struct Cell
{
  TypeDonnee donnee[30];
  struct Cell *fg, *fd;
} TypeNoeud;
/*Fonction et déclarations nécessaires pour le traitement
des différentes parties de l'exercice*/
typedef struct Cell2
{
  TypeDonnee donnee[30];
  struct Cell2 *suivant;
} TypeCellule;
typedef TypeCellule *Pile;
Pile initialiser()
{
  return NULL;
}

char EstVide(Pile P)
{
  return (P == NULL) ? 1 : 0;
}

void Empiler(Pile * pP, TypeDonnee * elem)
{
  Pile q;
  q = (TypeCellule *) malloc(sizeof(TypeCellule));
  strcpy(q->donnee, elem);
  q->suivant = *pP;
  *pP = q;
}
char Depiler(Pile * pP, TypeDonnee * pelem)
{
  Pile q;
```

```
  if (EstVide(*pP))
    return 1;
  strcpy(pelem, (*pP)->donnee);
  q = *pP;
  *pP = (*pP)->suivant;
  free(q);
  return 0;
}

void Afficherpile(Pile P)
{
  Pile q;
  q = P;
  while (q != NULL)
    {
      printf("%s\t", q->donnee);
      q = q->suivant;
    }
  puts("");
}

void Detruire(Pile * pP)
{
  Pile q;
  while (*pP != NULL)
    {
      q = *pP;
      *pP = (*pP)->suivant;
      free(q);
    }
  *pP = NULL;
}

void Vider(Pile * pP)
{
  Detruire(pP);
  *pP = NULL;
}
void DetruireArbre(TypeNoeud ** pP)
{
  TypeNoeud *qfg, *qfd;
  if (*pP != NULL)
    {
      qfg = (*pP)->fg;
```

```
      qfd = (*pP)->fd;
      free(*pP);
      DetruireArbre(&qfg);
      DetruireArbre(&qfd);
    }
}
void Traiter(TypeNoeud * noeud)
{
  printf("%s, ", noeud->donnee);
}
void ParcoursPrefixe(TypeNoeud * racine)
{
  if (racine != NULL)
    {
      Traiter(racine);
      ParcoursPrefixe(racine->fg);
      ParcoursPrefixe(racine->fd);
    }
}
```

b)

```
void Evaluation(TypeNoeud * racine, Pile * P)
{
  char a[30], b[30], c[30];
  if (racine != NULL)
    {
      Evaluation(racine->fd, P);
      Evaluation(racine->fg, P);
      if (strcmp(racine->donnee, "+") == 0)
        {
          Depiler(P, a);
          Depiler(P, b);
          sprintf(c, "%d", atoi(a) + atoi(b));
          Empiler(P, c);
        }
      else if (strcmp(racine->donnee, "*") == 0)
        {
          Depiler(P, a);
          Depiler(P, b);
          sprintf(c, "%d", atoi(a) * atoi(b));
          Empiler(P, c);
        }
      else if (strcmp(racine->donnee, "-") == 0)
        {
          Depiler(P, a);
```

```
            Depiler(P, b);
            sprintf(c, "%d", atoi(a) - atoi(b));
            Empiler(P, c);
          }
        else if (strcmp(racine->donnee, "/") == 0)
          {
            Depiler(P, a);
            Depiler(P, b);
            sprintf(c, "%d", atoi(a) / atoi(b));
            Empiler(P, c);
          }
        else
          Empiler(P, racine->donnee);
    }
}
```

c)

```
void GenereExpPrefixee(TypeNoeud * racine, Pile * expression)
{
  if (racine != NULL)
    {
      GenereExpPrefixee(racine->fd, expression);
      GenereExpPrefixee(racine->fg, expression);
      Empiler(expression, racine->donnee);
    }
}
```

d)

```
void CreationArbreDePrefixe(TypeNoeud * racine, Pile * prefix)
{
  TypeDonnee *elem;
  TypeNoeud *Nouveau;
  if (prefix != NULL)
    {
      Depiler(prefix, elem);
      strcpy(racine->donnee, elem);
      if (strcmp(elem, "+") == 0 || strcmp(elem, "-") == 0 ||
          strcmp(elem, "*") == 0 || strcmp(elem, "/") == 0)
        {
          racine->fg = (TypeNoeud *) malloc(sizeof(TypeNoeud));
          racine->fd = (TypeNoeud *) malloc(sizeof(TypeNoeud));
          CreationArbreDePrefixe(racine->fg, prefix);
          CreationArbreDePrefixe(racine->fd, prefix);
        }
      else
```

```
        {
          racine->fg = NULL;
          racine->fd = NULL;
        }
    }
}
```

e)

```
void ExprParenthese(TypeNoeud * racine, Pile * Cumul)
{
  Pile q;
  char a[30], b[30], c[30];
  if (racine != NULL)
    {
      ExprParenthese(racine->fd, Cumul);
      ExprParenthese(racine->fg, Cumul);
      if (strcmp(racine->donnee, "+") == 0
          || strcmp(racine->donnee, "-") == 0
          || strcmp(racine->donnee, "*") == 0
          || strcmp(racine->donnee, "/") == 0)
        {
          Depiler(Cumul, a);
          Depiler(Cumul, b);
          strcpy(c, "(");
          strcat(c, a);
          strcat(c, racine->donnee);
          strcat(c, b);
          strcat(c, ")");
          Empiler(Cumul, c);
        }
      else
        Empiler(Cumul, racine->donnee);
    }
}
```

f)

```
void ParentheseEnPrefixe(Pile * Original, Pile * Cumul, Pile * final)
{
  Pile q;
  char a[20];
  q = *Original;
  if (q != NULL)
    {
      if (strcmp(q->donnee, "+") == 0 || strcmp(q->donnee, "-") == 0
```

```
          || strcmp(q->donnee, "*") == 0 || strcmp(q->donnee, "/") == 0)
        {
          Empiler(Cumul, q->donnee);
        }
      else if (strcmp(q->donnee, ")") == 0)
        {
        }
      else if (strcmp(q->donnee, "(") == 0)
        {
          Depiler(Cumul, a);
          Empiler(final, a);
        }
      else
        {
          Empiler(final, q->donnee);
        }
      q = q->suivant;
      return ParentheseEnPrefixe(&q, Cumul, final);
    }
}
/*Dans cette fonction, l'exp parenthésée est
écrite sous format préfixée et puis l'arbre est construit*/
void CreationArbredeParenthese(TypeNoeud * racine, Pile * exprparenthese)
{
  Pile Cumul, prefix;
  Cumul = initialiser();
  prefix = initialiser();
  ParentheseEnPrefixe(exprparenthese, &Cumul, &prefix);
  CreationArbreDePrefixe(racine, &prefix);
  Detruire(&Cumul);
  Detruire(&prefix);
}
```

Le programme principal

```
int main()
{
  Pile P = NULL;
  char car[20] = "\0";
  P = initialiser();
  TypeNoeud *racine;
/* Saisir l'expression parenthésée entrée par l'utilisateur
l'exp est considérée complètement parenthésée ex ((10+2)/3) */
  printf("Entrez les membres de l'expr. préfixée en
    appuyant sur entrer à chaque fois qu'un
```

```
    élement est saisi et tapez terminer pour finir\n");
  scanf("%s", car);
  while (strcmp(car, "terminer") != 0)
    {
      Empiler(&P, car);
      scanf("%s", car);
    }
  racine = (TypeNoeud *) malloc(sizeof(TypeNoeud));
/* (d) ou (f) (meme principe) */
/* Créer l'arbre à partir de l'expression parenthésée */
  CreationArbredeParenthese(racine, &P);
  puts("L'arbre créé est le suivant");
  ParcoursPrefixe(racine);
  puts("");
  Vider(&P);
/* (c) Générer l'expression préfixée à partir de l'arbre,
le résultat se trouve dans la pile "P" */
  GenereExpPrefixee(racine, &P);
  printf("L'expression préfixée à partir de l'arbre= \n");
  Afficherpile(P);
  puts("");
  Vider(&P);
/* (b) Évaluation de l'exp. arith., le résultat se trouve dans
 la pile "P" contenant un seul élément à la sortie de la fonction */
  Evaluation(racine, &P);
  printf("Evaluation de l'exp arith. =%s\n", P->donnee);
  Vider(&P);
/* (e) L'expression parenthésée est générée à partir de l'arbre
le résultat se trouve dans la pile "P" contenant un seul élément
à la sortie de la fonction */
  ExprParenthese(racine, &P);
  printf("Exp. parenthésée générer de l'arbre=%s\n", P->donnee);
  Vider(&P);
  DetruireArbre(&racine);
  return 0;
}
```

23.5

a)

```
typedef struct noeud{
  char lettre;  /* représente le caractère alphabétique */
  struct noeud *fg, *fd; /* fils gauche et fils droit */
}Noeud, *Dictionnaire;
```

b)

```
int EstDansLeDicoRecurs(Dictionnaire dico, char *mot)
{
  Dictionnaire p = dico;
  /* on cherche la première lettre du mot */
  while (p!=NULL && mot[0]!=p->lettre)
    p = p->fd;
  if (p==NULL) /* lettre non trouvée */
    return 0;
  if (mot[0]=='\0') /* lettre trouvée et fin du mot */
    return 1;
  /* appel récursif. On avance d'une lettre */
  /* à la fois dans le mot et dans le dico */
  return EstDansLeDicoRecurs(p->fg, &mot[1]);
}
```

c)

```
int EstDansLeDicoIter(Dictionnaire dico, char *mot)
{
  Dictionnaire p = dico;
  int i=0, trouve=0;
  while (p!= NULL)
    {
      if (p->lettre != mot[i])
        p = p->fd;  /* on cherche une autre lettre possible */
      else /* la lettre a été trouvée */
        {
          if (mot[i] == '\0') /* le mot complet est trouvé */
            {
              trouve = 1;
              p = NULL; /* on sort de la boucle */
            }
          else
            {
              i++; /* on passe à la lettre suivante du mot */
              p = p->fg;
            }
        }
    }
  return trouve;
}
```

d)

```
void VerifieOrthographe(Dictionnaire dico, char *fichierTexte)
{
  char mot[150];
  FILE *fp;
  if ((fp = fopen(fichierTexte, "r"))==NULL)
    {
      printf("Erreur d'ouverture de fichier\n");
      return;
    }
  while (fscanf(fp, "%s", mot) == 1)
    if (!EstDansLeDicoIter(dico, mot))
      puts(mot);
}
```

e)

```
/* créer un dictionnaire contenant un seul mot */
Dictionnaire CreeDico_1mot(char * mot)
{
  int i;
  Dictionnaire dico=NULL, nouveau;
  /* pour chaque lettre y compris le '\0' */
  /* on parcourt le mot à l'envers et on */
  /* fait les insertions en tête de liste */
  for (i=strlen(mot)+1 ; i>=0 ; i--)
    { /* allocation d'un noeud */
      nouveau = (Dictionnaire)malloc(sizeof(Noeud));
      nouveau->lettre = mot[i];
      nouveau->fg = dico; /* chaînage */
      nouveau->fd = NULL;
      dico = nouveau; /* nouvelle tête de liste */
    }
  return dico; /* on retourne la tête de liste */
}
/* ajouter un mot dans un dictionnaire */
Dictionnaire AjouterMot(Dictionnaire dico, char *mot)
{
  Dictionnaire p = dico;
  /* on cherche la première lettre du mot */
  while (p!=NULL && mot[0]!=p->lettre)
    p = p->fd;
  if (p==NULL) /* lettre non trouvée, ajout du mot */
    {
      /* création d'une liste de nœuds contenant mot */
      Dictionnaire nouveau = CreeDico_1mot(mot);
```

```
        /* insertion dans l'ancien arbre */
        nouveau->fd = dico;
        return nouveau; /* nouvelle racine */
      }
    if (mot[0]=='\0') /* le mot est déjà dans le dico */
      return dico;
    /* appel récursif. On avance d'une lettre */
    /* à la fois dans le mot et dans le dico */
    p->fg = AjouterMot(p->fg, &mot[1]);
    return dico;
}
```

f)

```
void SauvegardeDicoRecurs(FILE *fpw, Dictionnaire dico,
                          char *debutMot, int i)
{
    if (dico !=NULL)
      {
        debutMot[i] = dico->lettre;
        if (debutMot[i] == '\0')
          fprintf(fpw, "%s\n", debutMot);
        SauvegardeDicoRecurs(fpw, dico->fg, debutMot, i+1);
        SauvegardeDicoRecurs(fpw, dico->fd, debutMot, i);
      }
}
void SauvegardeDico(char *fichierDico, Dictionnaire dico)
{
    FILE *fpw;
    char mot[150];
    if ((fpw = fopen(fichierDico, "w"))==NULL)
      {
        printf("Erreur d'ouverture de fichier\n");
        printf("Répertoire inexistant ou droits insuffisants\n");
        return;
      }
    SauvegardeDicoRecurs(fpw, dico, mot, 0);
    fclose(fpw);
}
```

g)

```
Dictionnaire ChargementDico(char *fichierDico)
{
    FILE *fpr;
    char mot[150];
```

```
  Dictionnaire dico=NULL;
  if ((fpr = fopen(fichierDico, "r"))==NULL)
    {
      printf("Erreur d'ouverture de fichier\n");
      printf("Fichier inexistant ou droits insuffisants\n");
      return NULL;
    }
  while (fscanf(fpr, "%s", mot) == 1)
    dico = AjouterMot(dico, mot);
  fclose(fpr);
  return dico;
}
```

h) Pour que les mots du dictionnaire soient rangés dans l'ordre alphabétique lors de la sauvegarde, il suffit que, dans l'arbre, les lettres soient rangées dans l'ordre alphabétique lors du parcours des listes fils droit. Autrement dit, il suffit que le code ASCII de chaque noeud soit plus petit que le code ASCII de son fils droit. (sachant que le code ASCII de '\0' est 0.)

Graphes 24

24.1 Définition mathématique d'un graphe

Intuitivement, un graphe G est un ensemble de sommets reliés par des arcs (voir la figure 24.1 où les sommets sont représentés par des cercles numérotés et les arcs sont représentés par des flèches). Un graphe peut représenter un réseau routier (avec éventuellement des sens uniques), un réseau de transport aérien, un réseau informatique, ou encore des choses plus abstraites.

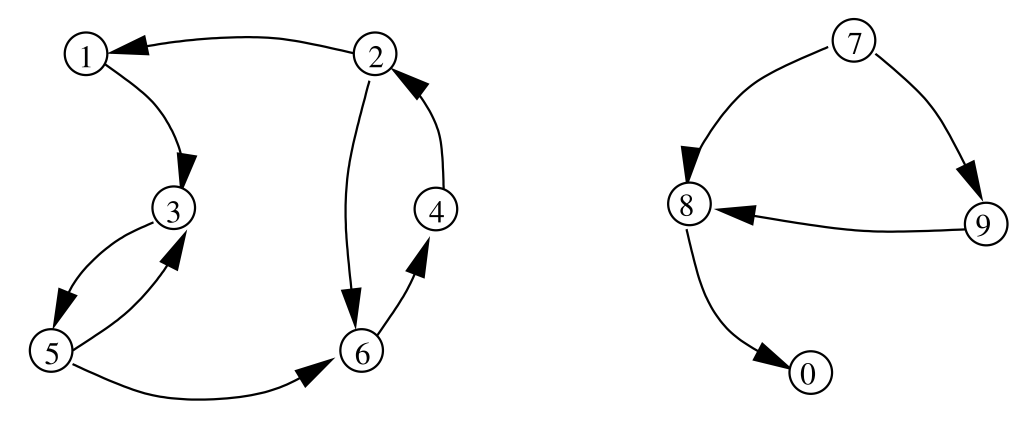


Figure 24.1– **Exemple de graphe (orienté)**

Mathématiquement, les sommets forment un ensemble $S = \{s_0, s_1, \dots, s_{n-1}\}$. Ici, les sommets sont numérotés de 0 à $n - 1$, et n est le nombre de sommets du graphe. De plus, chaque arc est un couple (i, j) de sommets, ce qui signifie que l'arc se trouve entre le sommet s_i et le sommet s_j. Le graphe G est finalement formé de l'ensemble des sommets, et d'un ensemble d'arcs.



Les couples qui constituent les arcs sont orientés. Par exemple, il peut y avoir un arc (i, j) du sommet s_i vers le sommet s_j, alors qu'il n'y a pas d'arc (j, i) du sommet s_j vers le sommet s_i

Un sommet s_j tel qu'il existe un arc (i, j) est appelé *successeur de* s_i.

24.2 Chemins dans un graphe

Un *chemin* dans un graphe G est une suite de sommets, tels que deux sommets consécutifs du chemin sont liés par un arc (orienté).

Par exemple, dans le graphe de la figure 24.1, la suite $(4, 2, 1, 3, 5)$ est un chemin. Par contre, la suite $(4, 2, 3, 5)$ n'est pas un chemin car il n'y a pas d'arc de 2 vers 3.

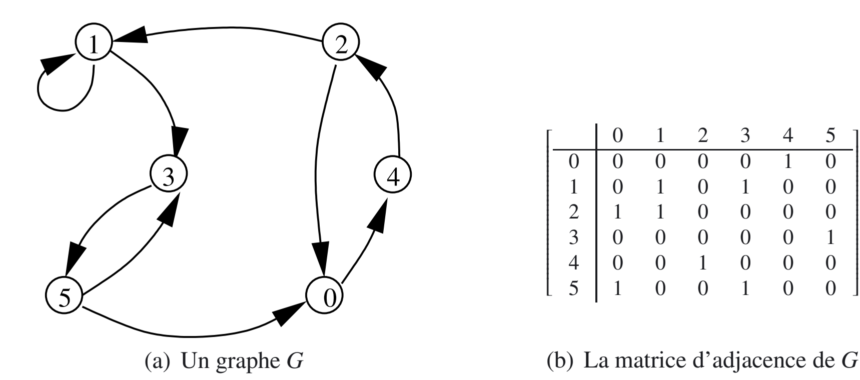
En d'autres termes, le chemin est une suite $(i_0, \dots, i_k)$ de sommets du graphe telle qu'il y ait un arc (i_{a-1}, i_a) pour chaque $a = 1, \dots, k$.

Soient deux sommets s et t dans un graphe G. Un chemin dans G de s à t est un chemin dans G dont le premier sommet est le sommet s et dont le dernier sommet est le somme t.

Par exemple, dans le graphe de la figure 24.1, la suite $(4, 2, 1, 3, 5)$ est un chemin de 4 à 5. Par contre, il n'existe pas de chemin de 4 à 0.

24.3 Représentation par matrices d'adjacence

On souhaite définir une structure de données pour représenter un graphe. Considérons $S = \{s_0, s_1, \dots, s_{n-1}\}$ l'ensemble des sommets. Nous allons représenter le graphe par une matrice A (ou encore, A est un tableau à double entrée, voir le chapitre 15) de taille $n \times n$. L'élément $A[i][j]$ (intersection de la ligne i et de la colonne j dans la matrice) vaudra 1 s'il y a un arc (i, j) du sommet s_i vers le sommet s_j, et $A[i][j]$ vaudra 0 sinon (voir la figure 24.2).



	0	1	2	3	4	5
0	0	0	0	0	1	0
1	0	1	0	1	0	0
2	1	1	0	0	0	0
3	0	0	0	0	0	1
4	0	0	1	0	0	0
5	1	0	0	1	0	0

(a) Un graphe G (b) La matrice d'adjacence de G

Figure 24.2 – La matrice d'adjacence d'un graphe

Du point de vue structures de données, on peut mettre les données du graphe dans une structure `Graphe`.

```
typedef struct
   {
      /* mettre ici les données relatives aux sommets */
      /* (nom, numéro, etc. du sommet) */
   }Sommet;
```

```
typedef struct
   {
      int n;  /* nombre de sommets */
      Sommet *tabSomm;  /* tableau de sommets */
      char **matrice;  /* les éléments valent 0 ou 1 */
   }Graphe;
```

Exercices

24.1 (∗∗) La configuration matérielle du réseau informatique intranet d'une entreprise nationale est enregistrée dans un fichier texte. Le fichier contient :

- Le nombre n de noeuds du réseau (chaque noeud correspondant plus ou moins à une localisation géographique) ;
- Le nombre m de connexions entre noeuds (une connexion correspondant à un câble). On supposera qu'entre deux noeuds il y a au plus 1 câble.
- La liste des n noms des noeuds ;
- La liste des m connexions, chaque connexion étant représentée par les numéros des deux noeuds extrémités.

a) Proposez une structure de données pour représenter le réseau en mémoire centrale. On rassemblera toutes les données nécessaires dans une structure `Reseau`.

b) Écrire une fonction de chargement du réseau en mémoire.

c) Écrire une fonction de sauvegarde du réseau stocké en mémoire.

d) Écrire une fonction qui détermine si deux machines données sont connectées par un câble.

e) Qu'est-ce qu'un chemin dans le réseau ?

f) Écrire une fonction qui prend en paramètre un tableau de numéros de noeuds du réseau et qui détermine si chaque noeud du tableau est connecté au suivant par un câble.

24.2 (∗) **(graphes non orientés)** On appelle *graphe non orienté* un graphe dont toutes les arêtes vont dans les deux sens. Autrement dit, dans un graphe non orienté, s'il y a un arc d'un sommet s_i vers le sommet s_j, alors il y a aussi un arc du sommet s_j vers le sommet s_j. La paire $\{i, j\}$ est alors appelée une *arête* de G.

Soit un graphe donné par une matrice d'adjacence. Donner une propriété de la matrice qui soit caractéristique d'un graphe non orienté. Donner un algorithme qui prend en paramètre un graphe donné sous forme de matrice d'adjacence, et qui renvoie 1 si le graphe est non orienté, et 0 sinon.

24.3 (∗) Soit un graphe G à n sommets. Un *coloriage de G* est une fonction qui à chaque sommet de G associe un entier appelé couleur du sommet. On représente un coloriage de G par un tableau de n entiers.

Un coloriage de G est dit *correct* si pour tout arc (i, j) de G la couleur de s_i est différente de la couleur de s_j.

Écrire une fonction qui prend en paramètre un graphe G représenté sous forme de matrice d'adjacence et un coloriage, et qui renvoie 1 si le coloriage de G est correct et 0 sinon.

Corrigés

24.1

a)

```
typedef struct noeud
{
  int noeud1;
  int noeud2;
} TypeConnexe;
typedef struct Cell
{
  int n;
  int m;
  char **nom;
  TypeConnexe *noeudconnexe;
} TypeReseau;
```

b)

```
/*On suppose dans cet exercice que les noms des machines
 sont rangées dans l'ordre de leur numéro nom1 correspond
 au sommet numero 1 ...*/
TypeReseau Chargement(char *nomfichier)
{
  int i;
  FILE *fp;
  char tmp[100];
  TypeReseau reseau;
```

```
  if ((fp = fopen(nomfichier, "rt")) == NULL)
    {
      puts("Erreur d'ouverture du fichier");
      exit(1);
    }
  fscanf(fp, "%d", &reseau.n);
  fscanf(fp, "%d", &reseau.m);
  reseau.nom = (char **) malloc(reseau.n * sizeof(char *));
  for (i = 0; i < reseau.n; i++)
    {
      fscanf(fp, "%s", tmp);
      reseau.nom[i] = (char *) calloc(strlen(tmp) + 1, sizeof(char));
      strcpy(reseau.nom[i], tmp);
    }
  reseau.noeudconnexe = (TypeConnexe *) malloc(sizeof(TypeConnexe));
  for (i = 0; i < reseau.m; i++)
    fscanf(fp, "%d %d", &reseau.noeudconnexe[i].noeud1,
           &reseau.noeudconnexe[i].noeud2);
  fclose(fp);
  return reseau;
}
```

c)

```
void Sauvegarde(char *nomfichier, TypeReseau reseau)
{
  FILE *fp;
  int i;
  if ((fp = fopen(nomfichier, "wt")) == NULL)
    {
      puts("Erreur d'ouverture du fichier");
      exit(1);
    }
  fprintf(fp, "%d\n", reseau.n);
  fprintf(fp, "%d\n", reseau.m);
  for (i = 0; i < reseau.n; i++)
    fprintf(fp, "%s\n", reseau.nom[i]);
  for (i = 0; i < reseau.m; i++)
    fprintf(fp, "%d %d\n", reseau.noeudconnexe[i].noeud1,
            reseau.noeudconnexe[i].noeud2);
  fclose(fp);
}
```

d)

```
int Connecte(int machine1, int machine2, TypeReseau reseau)
{
```

```
  int i;
  for (i = 0; i < reseau.m; i++)
    {
      if (reseau.noeudconnexe[i].noeud1 == machine1)
        {
          if (reseau.noeudconnexe[i].noeud2 == machine2)
            return 1;
          else
            return 0;        /*entre 2 noeuds au plus un cable */
        }
    }
  return 0;
}
```

e) Un chemin dans le réseau est une suite de sommets tel qu'il existe un câble de chaque sommet vers le sommet suivant dans le chemin.

d)

```
int Tabconnecte(int *tab, int n, TypeReseau reseau)
{
  int i, flag;
  for (i = 0; i < n - 1; i++)
    {
      flag = Connecte(tab[i], tab[i + 1], reseau);
      if (flag == 0)
        return 0;
    }
  return 1;
}
```

24.2

```
/*Dans un graphe non orienté, la matrice d'adjacence est
symétrique aij=aji et donc elle est la même en transposée*/
int GrapheOrientee(int **matadjacence, int n)
{
  int i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (matadjacence[i][j] != matadjacence[j][i])
        return 0;
  return 1;
}
```

24.3

```
int GrapheEtColoriage(int **matadjacence, int *coloriage, int n)
{
  int i, j;
  for (i = 0; i < n; i++)
    {
      for (j = 0; j < n; j++)
        if (matadjacence[i][j] == 1)
          if (coloriage[i] != coloriage[j])
            return 0;
    }
  return 1;
}
```

PARCOURS DE GRAPHES 25

Les parcours de graphes permettent de visiter les sommets du graphe à partir d'un sommet de départ en suivant les arcs du graphe. Les sommets visités sont les sommets qui sont accessibles à partir du sommet de départ par un chemin dans le graphe.

25.1 PARCOURS EN PROFONDEUR RÉCURSIF

Lors du parcours d'un graphe, on marque les sommets avec une étiquette 0 ou 1 pour indiquer les sommets qui ont déjà été visités. Cela évite de passer plusieurs fois par le même sommet et assure que le programme ne boucle pas.

Le parcours en profondeur commence par marquer tous les sommets à 0 (sommets non encore visités).

Le principe du parcours en profondeur récursif est de créer une fonction récursive de visite des sommets. La visite d'un sommet *V* consiste à marquer *V* à 1, puis à visiter tous les successeurs de *V* qui n'ont pas été précédemment visités tour à tour (on reconnaît les sommets qui n'ont pas été visités car ils sont marqués à 0). À la fin de l'algorithme, les sommets qui sont accessibles à partir du sommet de départ par un chemin sont marqués à 1 ; tous les autres sommets sont marqués à 0.
Voici un schéma d'algorithme de parcours en profondeur récursif.

```
Algorithme
en entrée : sommet de départ D;
            un graphe G;
début
   marquer tous les sommets à 0;
   Visiter(G, D);  /* D est le sommet de départ */
fin

PROCEDURE Visiter(Graphe G, Sommet V)
début
   marquer le sommet V à 1;
   pour chaque successeur S de V  faire
          si S est marqué à 0  faire   /* condition d'arret */
              Visiter(G, S);
fin
```

En pratique, pour marquer les sommets à 0 ou à 1, on rajoute une donnée (par exemple un `int`) dans la structure `Sommet` (voir chapitre 24).

Propriété
Les sommets visités par le parcours en profondeur sont les sommets *V* tels qu'il existe un chemin du sommet de départ *D* au sommet *V*.

25.2 Parcours en largeur

Le parcours en largeur permet, en utilisant une file, de parcourir les sommets d'un graphe à partir d'un sommet de départ *D*, du plus proche au plus éloigné de *D*. On visite les voisins de *D*, puis les voisins des voisins de *D*, etc.

Le principe est le suivant. Lorsqu'on visite un sommet, on insère ses voisins non encore insérés (marqués à 0) dans une file, et on les marque à 1. On défile les sommets lorsqu'on les visite. On itère le processus tant que la file est non vide. À mesure qu'on insère les sommets dans la file, on les marque pour ne pas les insérer deux fois.
Le schéma de l'algorithme est le suivant :

```
Algorithme
début
   initialiser une file F à vide;
   marquer tous les sommets à zéro;
   marquer le sommet de départ D à 1;
   insérer D dans F;
   tant que F est non vide faire
      début
         défiler un sommet V de F;
         pour chaque voisin S de V
            si S est marqué à 0  faire
               début
                  marquer S à 1;
                  insérer S dans F;
               fin
      fin
fin
```

L'intérêt du parcours en largeur est que les sommets sont visités dans l'ordre des distances croissantes au sommet de départ. Avec un peu d'astuce, on peut connaître le **plus court chemin d'un sommet à un autre**, c'est-à-dire le chemin comportant le moins d'arcs possible (voir les exercices 5 et 6).

Exercices

25.1 (∗) Appliquer le parcours en profondeur au graphe de la figure 25.1 en prenant pour sommet de départ le sommet 2, puis en prenant pour sommet de départ le sommet 7. On donnera la liste des sommets visités dans l'ordre où ils sont visités.

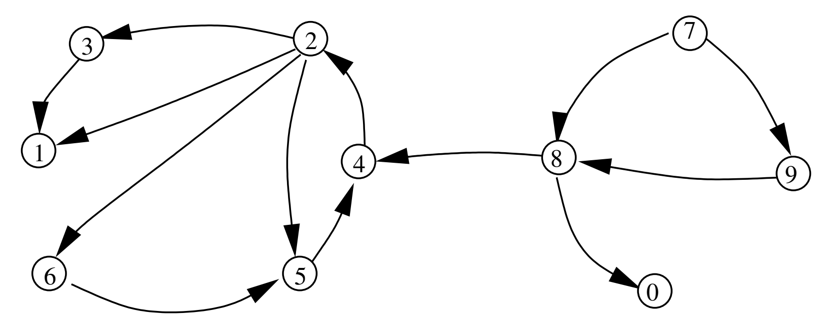


Figure 25.1 – Exemple de graphe

25.2 (∗∗)

a) Implémenter le parcours en profondeur récursif avec une représentation du graphe sous forme de matrice d'adjacence.

b) Écrire une fonction qui prend en paramètre deux sommets s_1 et s_2 dans un graphe donné sous forme de martrice d'adjacence, et qui renvoie 1 s'il existe un chemin de s_1 à s_2 et renvoie 0 sinon.

25.3 (∗) Appliquer le parcours en largeur au graphe de la figure 25.2 en prenant pour sommet de départ le sommet 4, puis en prenant pour sommet de départ le sommet 7. On donnera la liste des sommets visités dans l'ordre où ils sont visités et on maintiendra à jour une file.

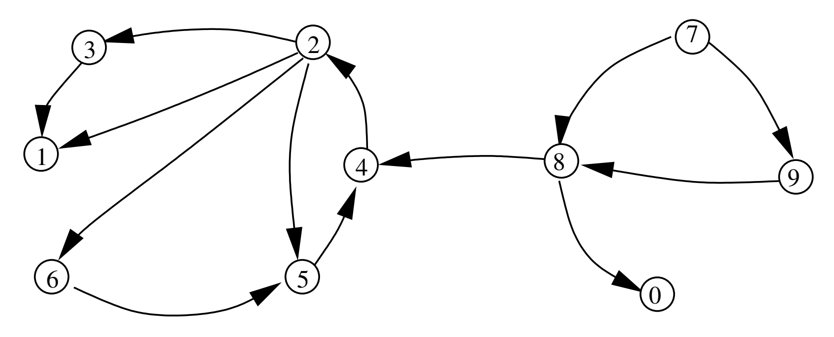


Figure 25.2 – Exemple de graphe

25.4 (∗) Donner une implémentation en *C* du parcours en largeur.

25.5 (∗) **(plus court chemin)** Appliquer le parcours en largeur au graphe de la figure 25.3 en prenant pour sommet de départ le sommet 4, puis en prenant pour sommet de départ le sommet 8. On donnera la liste des sommets visités dans l'ordre où ils sont visités et on maintiendra à jour une file.

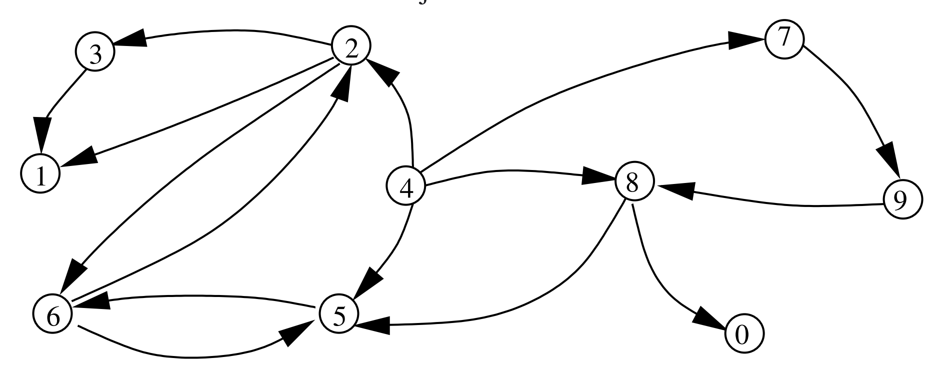


Figure 25.3 – Exemple de graphe

Pour chaque sommet S inséré dans la file, on dessinera une flèche de couleur du sommet S vers le dernier sommet V défilé. Que remarque-t'on en suivant les flèches de couleur ?

25.6 (∗∗) **(implémentation du plus court chemin)** Écrire une fonction *C* qui donne le plus court chemin entre deux sommets s_1 et s_2 d'un graphe donné sous forme de matrices d'adjacence. Pour chaque sommet S inséré dans la file, on mémorisera le numéro `preced` du dernier sommet V défilé avant d'insérer S dans la file. Pour afficher le chemin, on suivra les numéros `preced` à la manière des pointeurs `suivant` dans un parcours de liste chaînée.

Corrigés

25.1

On choisit d'ordonner les sommets en utilisant l'ordre trigonométrique. Les successeurs du sommet 2 sont donc traités dans l'ordre 3 puis 1 puis 6 puis 5. De même pour le sommet 7, ses successeurs sont dans l'ordre 8 puis 9.

- sommet **2** : 2 ; 3 ; 1 ; 6 ; 5 ; 4
- sommet **7** : 7 ; 8 ; 4 ; 2 ; 3 ; 1 ; 6 ; 5 ; 0 ; 9

25.2

a)

```
typedef struct
{
  /* éléments descriptif du sommet... en fonction de l'application */
  int marque;
} Sommet;
typedef struct
{
  int n;
  Sommet *tabSomm;
  char **matrice;
} Graphe;
void visiter(Graphe G, int sommet)
{
  int i;
  // Marquer sommet
  G.tabSomm[sommet].marque = 1;
  // Parcourir les successeurs
  for (i = 0; i < G.n; i++)
    {
      if ((G.matrice[sommet][i] == 1) && (G.tabSomm[i].marque == 0))
        visiter(G, i);
    }
}

/* G est le graphe, et sommet est le numéro du sommet
   dans le tableau de sommet de G */
void profondeur(Graphe G, int sommet)
{
  /* Marquer tous les sommets à zéro */
  int i;
  for (i = 0; i < G.n; i++)
    G.tabSomm[i].marque = 0;
  /* Appel de visiter recursivement */
  visiter(G, sommet);
}
```

b)

```
int visiterChemin(Graphe G, int sommet, int s2)
{
  int i, result;
  // Marquer sommet
  G.tabSomm[sommet].marque = 1;
```

```
  // Sortir 1 si s2 est atteint
  if (sommet == s2)
    return 1;
  // Parcourir les successeurs
  result = 0;
  for (i = 0; i < G.n; i++)
    {
      if ((G.matrice[sommet][i] == 1) && (G.tabSomm[i].marque == 0))
        result = result || visiterChemin(G, i, s2);
    }
  return result;
}
int chemin(Graphe G, int s1, int s2)
{
  /* Marquer tous les sommets à zéro */
  int i;
  for (i = 0; i < G.n; i++)
    G.tabSomm[i].marque = 0;
  /* Appel de visiter recursivement */
  return visiterChemin(G, s1, s2);
}
```

25.3

On utilise comme l'ordonnancement des sommets l'ordre de leur numéro. Ainsi, 1 vient avant 2 qui vient avant 3 et ainsi de suite.

Détaillons le début du parcours en largeur depuis le sommet 4 :

1. tous les sommets sont marqués à 0 sauf 4 qui est marqué à 1,
2. la file F contient uniquement 4,
3. on défile 4 de F (qui devient vide),
4. 2 étant l'unique successeur de 4, on le marque à 1 et on l'enfile dans F,
5. on défile ensuite 2 de F et on regarde ses successeurs ; on trouve dans l'ordre 1, 3, 5 et 6 ; tous ses sommets sont marqués et on les enfile dans F,
6. F contient donc 6-5-3-1 et on défile un élément qui est donc 1 (le premier entré),
7. 1 n'a pas de successeur donc F n'est pas modifiée,
8. on défile un élément de F, on obtient donc 3,
9. 3 possède un successeur mais qui est marqué, F n'est pas pas modifié,
10. ...

On obtient ainsi comme résultats des parcours,

- depuis le sommet **4** : 4 - 2 - 1 - 3 - 5 - 6,
- depuis le sommet **7** : 7 - 8 - 9 - 0 - 4 - 2 - 1 - 3 - 5 - 6.

25.4

```
void largeur(Graphe G, int sommet)
{
  File F;
  int i, s;
  F = Initialiser();        /* voir chapitre 21 */
  for (i = 0; i < G.n; i++)
    G.tabSomm[i].marque = 0;
  G.tabSomm[sommet].marque = 1;
  Enfiler(&F, sommet);
  while (!EstVide(&F))
    {
      Defiler(&F, &s);
      for (i = 0; i < G.n; i++)
        {
          if ((G.matrice[s][i] == 1) && (G.tabSomm[i].marque == 0))
            {
              G.tabSomm[i].marque = 1;
            Enfiler(&F, s)}
        }
    }
}
```

25.5

Chaque fois que l'on dessine un arc, on le fait lorsqu'un sommet est atteint pour la première fois dans le parcours en largeur. Cela signifie que dans le parcours du graphe, le chemin qui mène à ce sommet et le plus court (en nombre d'arcs) depuis le sommet de départ du parcours. Conséquence, il suffit de *"remonter"* les flèches de couleur pour obtenir le plus court chemin arrivant à un sommet quelconque en partant du sommet de départ.

25.6

```
typedef struct
{
  /* éléments descriptif du sommet... en fonction de l'application */
  int marque;
  int preced;
} Sommet;
typedef struct
{
```

```
  int n;
  Sommet *tabSomm;
  char **matrice;
} Graphe;
void largeurmodifiee(Graphe G, int sommet)
{
  File F;
  int i, s;
  F = Initialiser();        /* voir chapitre 21 */
  for (i = 0; i < G.n; i++)
    {
      G.tabSomm[i].marque = 1;
      /* -1 signifie "pas de précédent" */
      G.tabSomm[i].preced = -1;
    }
  G.tabSomm[sommet].marque = 1;
  Enfiler(&F, sommet);
  while (!EstVide(&F))
    {
      Defiler(&F, &s);
      for (i = 0; i < G.n; i++)
        {
          if ((G.matrice[s][i] == 1) && (G.tabSomm[i].marque == 0))
            {
              G.tabSomm[i].marque = 1;
              G.tabSomm[i].preced = s;
            Enfiler(&F, s)}
        }
    }
}
void pluscourtchemin(Graphe G, int s1, int s2)
{
  largeurmodifiee(G, s1);
  printf("%d", s2);
  while (G.tabSomm[s2].preced != -1)
    {
      s2 = G.tabSomm[s2].preced;
      printf("->%d", s2);
    }
}
```

LISTES D'ADJACENCE 26

26.1 REPRÉSENTATION PAR LISTES D'ADJACENCE

La représentation des graphes par matrices d'adjacence présente un gros inconvénient : la taille du graphe en mémoire est proportionnelle au carré du nombre de sommets. Lorsqu'il y a un très grand nombre de sommets, cela engendre un coût qui peut devenir exhorbitant. Pour pallier à cet inconvénient, on indtroduit la représentation des graphes par listes d'adjacence. Cette représentation prend une taille mémoire linéaire en la somme du nombre d'arcs et du nombre de sommets.

Dans la représentation des graphes par listes d'adjacence, la liste des sommets du graphe est donnée sous forme de liste chaînée (voir figure 26.1). Pour représenter les arcs, on met une liste d'arcs dans chaque sommet. La liste des arcs issus d'un sommet s est une liste chaînée dont la tête de liste est mémorisée dans le sommet. Chaque arc contient un pointeur vers le sommet extremité de l'arc (en d'autres termes, un pointeur vers le successeur du sommet).
La structure de données de liste d'adjacence est (dans sa version la plus simple) la suivante :

```
typedef struct CellSommet
  {
    TypeDonnee donne; /* données du sommet (numéro, nom,...) */
    struct CellArc *liste_arcs; /* liste des arcs du sommet */
    struct CellSommet *suivant; /* pointeur vers sommet suivant*/
  }TypeSommet;

typedef struct CellArc
  {
    struct CellSommet *extremite; /* pointeur vers successeur */
    struct CellArc *suivant; /* pointeur vers arc suivant */
  }TypeArc;

/* le graphe est la liste des sommets : */
typedef TypeSommet *TypeGraphe;
```

Remarque

La structure `TypeSommet` contient un pointeur sur `TypeArc` et la structure `TypeArc` contient un pointeur sur `TypeSommet`. Le compilateur C permet cela en acceptant la déclaration d'un pointeur sur une structure alors que la structure n'est pas encore définie.

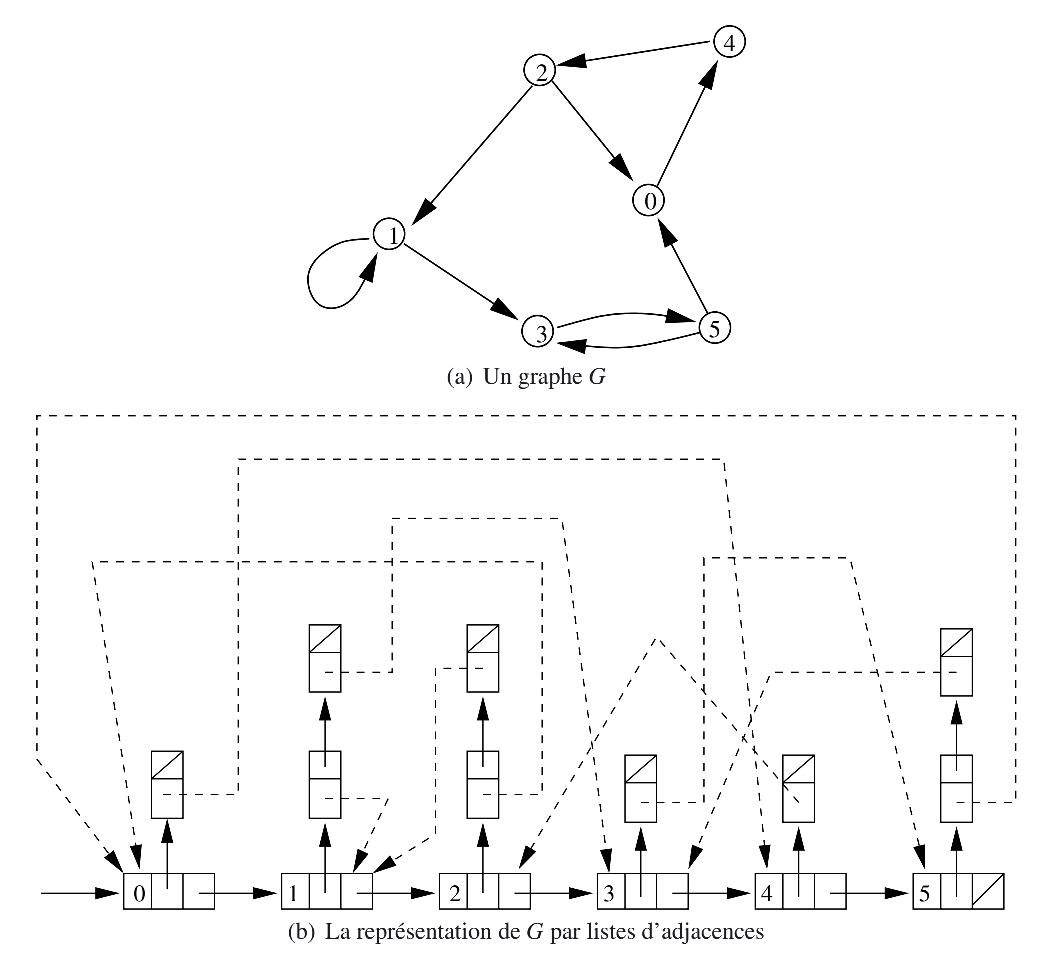


Figure 26.1– Les listes d'adjacence d'un graphe

Exercices

Pour les exercices de ce chapitre, on pourra utiliser les fonctions de gestion des listes chaînées du chapitre 19, notamment l'insertion en tête de liste et l'insertion en queue de liste.

26.1 (∗∗) Écrire un programme *C* qui construit la représentation sous forme de matrice d'adjacence d'un graphe donné sous forme de listes d'adjacence.

26.2 (∗ ∗ ∗) Écrire un programme *C* qui construit la représentation sous forme de listes d'adjacence d'un graphe donné sous forme de matrice d'adjacence.

26.3 (∗∗) La base de données d'une compagnie d'aviation est stockée dans un fichier texte au format suivant :

- La première ligne du fichier contient le nombre n d'aéroports du réseau.
- La deuxième ligne du fichier contient le nombre m d'avions de la semaine.
- Les n lignes suivantes du fichier contiennent les noms des n aéroports.
- Les m lignes suivantes du fichier contiennent la liste des vols de la semaine. Chaque vol est représenté par :
  - le numéro du vol ;
  - le jour et l'heure du décolage codée sur un `int` ;
  - le numéro de l'aéroport de départ du vol ;
  - le numéro de l'aéroport d'arrivée du vol.

a) Proposer une structure de données pour représenter la base de données en mémoire.

b) Écrire une fonction de chargement de la base de données en mémoire centrale.

c) Écrire une fonction qui affiche tous les vols de la semaine au départ d'un aéroport dont le nom est passé en paramètre.

d) Écrire une fonction qui détermine s'il y a un vol direct entre deux aéroports dont les noms sont passés en paramètre.

e) Écrire une fonction qui affiche le premier vol direct à partir d'une date courante entre deux aéroports passés en paramètre. La fonction doit afficher un message d'erreur s'il n'y a pas de vol entre ces deux villes.

26.4 (∗∗)

a) Implémenter le parcours en profondeur récursif avec une représentation du graphe sous forme de listes d'adjacence.

b) Écrire une fonction qui prend en paramètre deux sommets s_1 et s_2 dans un graphe donné sous forme de listes d'adjacence, et qui renvoie 1 s'il existe un chemin de s_1 à s_2 et renvoie 0 sinon.

26.5 (∗∗) Écrire les fonctions *C* permettant de faire le parcours en largeur d'un graphe donné sous forme de listes d'adjacence.

Corrigés

26.1

```
typedef struct
{
} Sommet;
typedef struct
{
  int n;
  Sommet *tabSomm;
  char **matrice;
} Graphe;
typedef struct CellSommet
{
  int numero;
  struct CellArc *liste_arcs;
  struct CellSommet *suivant;
} TypeSommet;
typedef struct CellArc
{
  struct CellSommet *extremite;
  struct CellArc *suivant;
} TypeArc;
typedef TypeSommet *TypeGraphe;
Graphe *liste2mat(TypeGraphe G)
{
  Graphe *Gp = calloc(1, sizeof(Graphe));
  /* Calcul de n */
  /* On utilise la partie recherche de la fonction
     InsereEnQueue du chapitre 19 */
  TypeSommet *temp = G;
  TypeArc *a;
  Gp->n = 1;
  while (temp->suivant != NULL)
    {
      temp = temp->suivant;
      Gp->n++;
    }
  /* Allocation de tabSomm et matrice
     voir les chapitres 12 et 15
     Partie laissée à la sagacité du lecteur.
   */
```

```
  /* Parcours du graphe G pour extraire les informations */
  temp = G;
  while (temp->suivant != NULL)
    {
      a = temp->liste_arcs;
      while (a != NULL)
        {
          Gp->matrice[temp->numero][a->extremite->numero] = 1;
          a = a->suivant;
        }
      temp = temp->suivant;
    }
  return Gp;
}
```

26.2

```
typedef struct
{
} Sommet;
typedef struct
{
  int n;
  Sommet *tabSomm;
  char **matrice;
} Graphe;
typedef struct CellSommet
{
  struct CellArc *liste_arcs;
  struct CellSommet *suivant;
} TypeSommet;
typedef struct CellArc
{
  struct CellSommet *extremite;
  struct CellArc *suivant;
} TypeArc;
typedef TypeSommet *TypeGraphe;
TypeGraphe mat2liste(Graphe * G)
{
  TypeGraphe Gp, temp, s;
  TypeArc *tmp;
  int i, j;
  /* CrÃľation de la liste des sommets */
  Gp = (TypeGraphe) calloc(1, sizeof(TypeSommet));
  temp = Gp;
```

```
  for (i = 1; i < G->n; i++)
    {
      temp->suivant = (TypeSommet *) calloc(1, sizeof(TypeSommet));
      temp = temp->suivant;
    }

  /* Conversion de la matrice en listes d'adjacence */
  temp = Gp;
  for (i = 0; i < G->n; i++, temp = temp->suivant)
    {
      s = Gp;
      for (j = 0; j < G->n; j++, s = s->suivant)
        {
          if (G->matrice[i][j] == 1)
            {
              if (temp->liste_arcs == NULL)
                {
                  temp->liste_arcs = (TypeArc *) calloc(1, sizeof(TypeArc));
                  tmp = temp->liste_arcs;
                  tmp->extremite = s;
                }
              else
                {
                  tmp->suivant = (TypeArc *) calloc(1, sizeof(TypeArc));
                  tmp->suivant->extremite = s;
                  tmp = tmp->suivant;
                }
            }
        }
    }
  return Gp;
}
```

26.3

a)

```
typedef struct CellSommet
{
  int aeroport;
  char *nom;
  struct CellArc *liste_arcs;
  struct CellSommet *suivant;
} TypeSommet;
typedef struct CellArc
```

```
{
  int numero_vol;
  int jour, heure;
  struct CellSommet *extremite;
  struct CellArc *suivant;
} TypeArc;
typedef TypeSommet *TypeGraphe;
```

b)

```
/*
  La lecture depuis un fichier a été présentée au chapitre 10
  nous ne la détaillerons pas ici. En revanche, un problème se pose
  avec les données. En effet, nous ne connaissons que le numéro
  des aéroports ce qui est insuffisant pour une liste chaînée.
  Il faut donc une fonction de recherche d'un sommet à partir
  de son numéro. Nous détaillons donc cette fonction, le reste
  de la correction étant laissé à la sagacité du lecteur.
  Nous supposons que le numéro passé à la fonction existe...
*/
TypeSommet *localise(TypeGraphe G, int numero)
{
  TypeGraphe tmp = G;
  while (tmp->aeroport != numero)
    tmp = tmp->suivant;
  return tmp;
}
```

c)

```
/*
Il faut utiliser la fonction Affiche du chapitre sur les listes
en utilisant la liste chaînée entre les sommets.
*/
```

d)

```
int voldirect(TypeGraphe G, int depart, int arrivee)
{
  TypeSommet *d, *a;
  TypeArc *tmp;
  d = localise(G, depart);
  a = localise(G, arrivee);
  tmp = d->liste_arcs;
  while ((tmp != NULL) && (strcmp(tmp->extremite->nom, a->nom) != 0))
    tmp = tmp->suivant;
  return (tmp != NULL);
}
```

e)

```
/*
La difficulté de cette question tient dans l'obligation de
trouver un vol après la date donnée en paramètre. Pour cela,
il faut modifier la fonction voldirect de la question
précédente afin de faire intervenir la date dans le test
d'existence.
*/
int apres(int j, int h, int jour, int heure)
{
  if (j < jour)
    return 0;
  if (jour < j)
    return 1;
  else
    return (h > heure);
}
int voldirectdate(TypeGraphe G, int depart, int arrivee, int jour, int heure)
{
  TypeSommet *d, *a;
  TypeArc *tmp;
  d = localise(G, depart);
  a = localise(G, arrivee);
  tmp = d->liste_arcs;
  while ((tmp != NULL) &&
         (strcmp(tmp->extremite->nom, a->nom) != 0) &&
         (apres(tmp->jour, tmp->heure, jour, heure)))
    tmp = tmp->suivant;
  return (tmp != NULL);
}
```

26.4

a)

```
typedef struct CellSommet
{
  int marque;
  struct CellArc *liste_arcs;
  struct CellSommet *suivant;
} TypeSommet;
typedef struct CellArc
{
  struct CellSommet *extremite;
  struct CellArc *suivant;
} TypeArc;
typedef TypeSommet *TypeGraphe;
void visiter(TypeSommet * s)
```

```
{
  s->marque = 1;
  TypeArc *voisin = s->liste_arcs;
  while ((voisin != NULL) && (voisin->extremite->marque != 1))
    {
      visiter(voisin->extremite);
      voisin = voisin->suivant;
    }
}
/* G est le graphe, et sommet est le numéro du sommet
   dans le tableau de sommets de G */
void profondeur(TypeGraphe G)
{
  /* Appel de visiter recursivement */
  visiter(G);
}
```

b)

```
/* voir les corrections du chapitre 25
 pour les modifications à effectuer. */
```

26.5

```
void largeur(TypeGraphe G, int sommet)
{
  File F;
  TypeSommet *s;
  TypeArc *voisins;
  F = Initialiser();          /* voir chapitre 21 */
  G->marque = 1;
  Enfiler(&F, G);
  while (!EstVide(&F))
    {
      Defiler(&F, s);
      voisins = s->liste_arcs;
      while (voisins != NULL)
        {
          if (voisins->extremite->marque != 1)
            {
              voisins->extremite->marque = 1;
              Enfiler(&F, voisins->extremite);
            }
          voisins = voisins->suivant;
        }
    }
}
```

Annexes

NOTIONS SUR LA COMPILATION



A.1 QU'EST-CE QU'UN COMPILATEUR *C ANSI* ?

Lorsqu'on écrit un programme, on écrit du code *C* dans un fichier texte. On donne toujours à ce fichier texte l'extension `.c` (de même qu'on utilise `.doc` pour des documents *Word* ou `.mp3` pour certains fichiers son, etc.). Le fichier `.c` s'appelle un *fichier source*. Il contient du *code source C*. Pour qu'un utilisateur (éventuellement le programmeur lui-même) puisse exécuter le programme, il faut créer un fichier *exécutable* à partir du fichier source. Pour créer un fichier exécutable, il faut utiliser un *compilateur*. Le compilateur va vérifier qu'il n'y a pas d'erreur de syntaxe dans le fichier source (comme l'oubli d'un point-virgule) et, s'il n'y a pas d'erreur, va générer le fichier exécutable. On peut ensuite lancer le programme exécutable, ce qui exécute le programme.

Les ordinateurs peuvent fonctionner avec différentes plate formes : *Windows*, *Linux*, *Mac OS*, *Unix*. Sous toutes ces plate formes, on trouve des compilateurs *C* ANSI qui peuvent être utilisés pour créer des programmes en langage *C ANSI*. Un compilateur *C ANSI* que l'on peut utiliser gratuitement (c'est un compilateur *open source*) est le compilateur `gcc`. Ce compilateur est disponible directement (éventuellement après installation) sous *Linux*, *Mac OS*, *Unix*. Pour utiliser `gcc` sous *Windows*, on peut télécharger et installer *Cygwin*, qui est un émulateur *Linux* qui marche sous *Windows* (ou encore Ubuntu Desktop Edition Wubc). Dans la suite de ce chapitre, on supposera que le lecteur est en mesure d'ouvrir une console (éventuellement sous *Cygwin*) et que *gcc* est installé. Sous *Linux*, pour faire apparaître une console ou un terminal (*shell*), cliquez sur l'icône représentant un écran d'ordinateur et un shell. Une fenêtre apparaît : c'est la console qui permet de commander le système *Linux*.

A.2 COMPILER SON PREMIER PROGRAMME

Dans la console, tapez `ls` (abréviation de *list*). Le contenu de votre répertoire d'accueil apparaît.

```
$ ls
```

Rappel

Un fichier représente des données sur le disque. Chaque fichier porte un nom. Les fichiers sont regroupés dans des répertoires. Chaque répertoire peut contenir des fichiers, ou d'autres répertoires qui contiennent eux aussi des fichiers...
Pour afficher le nom du répertoire courant, tapez `pwd`. (abréviation de *print working directory*)

A.2.1 Créer un répertoire

Pour créer un repertoire nommé “algorithmique” tapez :

```
$ mkdir algorithmique
```

(abréviation de *make directory*.)
Pour aller dans le répertoire `algorithmique` tapez :

```
$ cd algorithmique
```

(abréviation de *change directory*) Vérifiez par `pwd`.
Créez ensuite un répertoire “chapitre2” et allez dedans en utilisant `cd`.

A.2.2 Lancer un éditeur de texte

Pour écrire un programme, il faut taper le programme dans un éditeur de texte (*kate*, *xemacs*, *nedit*, *vi*, etc. à votre convenance). Par exemple, pour lancer l’éditeur *xemacs* sur le fichier `exercice1.c`, on tape :

```
$ xemacs exercice1.c &
```



Ne pas oublier l’extension `.c` pour les fichiers source. Cela pourrait provoquer la perte des données lors de la compilation

Dans l’éditeur , tapez un programme *C* qui affiche le message “bonjour” (voir chapitre 2). Pour sauvegarder, tapez (sous *xemacs*) `Ctrl-x Ctrl-s`. Le message “wrote exercice1.c” doit s’afficher en bas de la fenêtre. Dans la console, on peut vérifier en tapant `ls` que le fichier `.c` a bien été créé.

A.2.3 Compiler et exécuter le programme

Avant de compiler un programme pour le tester, il faut toujours sauver les dernières modifications. Pour pouvoir exécuter et tester un programme, il faut le *compiler*, ce qui génère un fichier exécutable à partir de votre code source. Pour cela, cliquez dans la console et tapez :

```
$ gcc exercice1.c -o exercice1
```

Deux cas peuvent se produire :

1. ou bien il y a un message d’erreur, qu’il faut essayer d’interpréter, qui indique une erreur dans votre code source (en précisant le numéro de la ligne de code où se trouve l’erreur). Corrigez l’erreur dans le programme *C* et recommencez... ;
2. ou bien il n’y a pas de message d’erreur et le compilateur a dû générer un fichier exécutable exercice1 (vérifiez le par `ls`).

Vous pouvez exécuter le programme en tapant le nom de l’exécutable :

```
$ ./exercice1
```

PROGRAMMATION MULTIFICHIERS

B

B.1 METTRE DU CODE DANS PLUSIEURS FICHIERS

Lorsqu'on écrit de longs programmes informatiques, il devient pénible, pour ne pas dire impossible, de mettre tout le code source dans un seul fichier. On crée alors plusieurs fichiers `.c` contenant chacun une ou plusieurs fonctions.

La difficulté vient du fait que, nécessairement, des fonctions de certains fichiers (par exemple le main) utilisent des fonctions qui sont définies dans d'autres fichiers. Pour cela, il est nécessaire de créer des fichiers d'en-tête, qui contiennent les définition des types (structures, etc.), et les prototypes de fonctions qui concernent plusieurs fichiers. Les fichiers d'en-tête, ou *header files*, ont une extension `.h`. Il faut les inclure dans les fichiers `.c` par une directive `#include`.

Exemple

Supposons qu'un `TypeArticle` regroupe les données d'un produit dans un magasin. La fonction `main`, dans le fichier `main.c`, appelle les fonctions `SaisitProduit` et `AfficheProduit`, qui sont définies dans le fichier `routines.c`. Les deux fichiers `.c` incluent le fichier `typeproduit.h`

```
/***************************************************\
|******* HEADER FILE  typeproduit.h ****************|
\**************************************************/

/* 1) Définition des structures et types */

typedef struct {
            int code;   /* code article */
            char denomination[100];  /* nom du produit */
            float prix; /* prix unitaire du produit */
            int stock;  /* stock disponible */
                 }TypeArticle;

/* 2) Prototypes des fonctions */

void SaisitProduit(TypeArticle *adr_prod);
void AfficheProduit(TypeArticle prod);
```

```
/**************************************************\
|********** SOURCE FILE  routines.c ***************|
\**************************************************/

#include <stdio.h>
#include "typeproduit.h"  /* attention aux guillemets */

void SaisitProduit(TypeArticle *adr_prod)
{
   printf("Code produit : ");
   scanf("%d", &adr_prod->code)
   printf("Dénomination : ");
   fgets(adr_prod->denomination, 100, stdin);
   printf("Prix : ");
   scanf("%f", &adr_prod->prix);
   printf("Stock disponible : ");
   scanf("%d", &adr_prod->stock);
}
void AfficheProduit(TypeArticle prod)
 {
   printf("Code : %d\n", prod.code);
   printf("Dénomination : %s\n", prod.denomination);
   printf("Prix : %f\n", prod.prix);
   printf("Stock disponible : %d\n", prod.stock);
}
```

```
/**************************************************\
|************* SOURCE FILE  main.c ****************|
\**************************************************/

#include "typeproduit.h"  /* attention aux guillemets */

int main(void)
{
   TypeProduit prod;
   SaisitProduit(&prod);
   AfficheProduit(prod);
}
```

B.2 Compiler un projet multifichiers

B.2.1 Sans makefile

Pour compiler l’exemple précédent sans `makefile` sous *Linux*, c’est à dire pour créer un fichier exécutable, il faut d’abord créer un fichier *objet* pour chaque fichier source.

```
$ gcc -c routines.c
$ gcc -c main.c
```

Ceci doit générer deux fichiers objets `routines.o` et `main.o`. On crée ensuite l'exécutable (par exemple appelé `produit.exe`) en réalisant l'édition des liens (*link*) par l'instruction suivante :

```
$ gcc routines.o main.o -o produit.exe
```

B.2.2 Avec `makefile`

Un `makefile` est un moyen qui permet d'automatiser la compilation d'un projet multifichier. Grace au `makefile`, la mise à jours des fichiers objets et du fichier exécutable suite à une modification d'un source se fait en utilisant simplement la commande :

```
$ make
```

Pour cela, il faut spécifier au système les dépendances entre les différents fichiers du projet, en créant un *fichier makefile*.

Pour l'exemple précédent, on crée un fichier texte de nom `makefile` contenant le code suivant :

```
produit.exe : routines.o main.o
	gcc routines.o main.o -o produit.exe
routines.o : routines.c typeproduit.h
	gcc -c routines.c
main.o: main.c typeproduit.h
	gcc -c main.c
```

Ce fichier comprend trois parties. Chaque partie exprime une règle de dépendance et une règle de reconstruction. Les règles de reconstruction (lignes 2, 4 et 6) commencent obligatoirement par une tabulation.

Les règles de dépendance sont les suivantes :

- Le fichier exécutable `produit.exe` dépend de tous les fichiers objets. Si l'un des fichier objets est modifié, il faut utiliser la règle de reconstruction pour faire l'édition des liens.
- Le fichier objet `routine.o` dépend du fichier `routine.c` et du fichier `typearticle.h`. Si l'un de ces deux fichiers est modifié, il faut utiliser la règle de reconstruction pour reconstruire `routine.o`
- De même, `main.o` dépend de `main.c` et `typearticle.h`.

COMPLÉMENTS SUR LE LANGAGE *C*



C.1 ÉNUMÉRATIONS

Une énumération est un type de données qui peut prendre un nombre fini de valeurs (2, 3,..., 15... valeurs différentes). Les différentes valeurs prises par les variables d'un type énumération sont des constantes.

Exemple 1.

Dans l'exemple suivant, le type `TypeAnimal` représente différentes sortes d'animaux. Toute variable de type `TypeAnimal` vaut soit `SINGE`, `CHIEN`, `ELEPHANT`, `CHAT` ou `GIRAFFE`.

```
#include <stdio.h>

/* Déclaration du type enum : */
 enum TypeAnimal {SINGE, CHIEN, ELEPHANT, CHAT, GIRAFFE};

/* Exemple de fonction utilisant ce type */
void AffichePropriete(TypeAnimal a)
{
   if (a == SINGE || a == ELEPHANT || a == GIRAFFE)
      puts("Animal sauvage");
   if (a == CHIEN || a == CHAT)
      puts("Animal de companie");
}
```

Exemple 2.

un type booléen. Le type booléen peut prendre deux valeurs : vrai ou faux. Ici, on impose que `VRAI` vaille 1 et `FAUX` vaille 0 dans la définition de l'énumération.

```
#include <stdio.h>

 enum Booleen {FAUX=0, VRAI=1}; /* Déclaration du type enum */

Booleen SaisieChoix(void)
{
   char choix;
   puts("Etes-vous d'accord ? (y/n)");
   choix = getchar();
   getchar();
```

```
   if (choix == 'y')
      return VRAI
   else
      return FAUX
}

void AfficheChoix(Booleen b)
{
   if (b)
      puts("Le client est d'accord");
   else
      puts("Le client n'est pas d'accord");
}

int main(void)
{
   Booleen b;
   b = SaisieChoix();
   AfficheChoix(b);
   return 0;
}
```

C.2 Unions

Une union est un type de données qui permet de stocker des données de types différents à la même adresse. Par exemple, on peut définir une union dont les éléments peuvent être soit `int` soit `float`. Dans tous les cas, la variable de ce type union prendra 4 octets en mémoire.

```
#include <stdio.h>

typedef  union data
   {
      int i;
      float x;
   }Data;

int main()
{
   Data d;
   int choix;
   puts("Voulez-vous entrer un entier (1) ou un réel (2) ?");
   scanf("%d", &choix);
   if (choix == 1)
```

```
      {
         scanf("%d", &d.i);
         printf("%d", d.i);
      }
   if (choix == 2)
      {
         scanf("%f", &d.x);
         printf("%f", d.x);
      }
   return 0;
}
```

Les types des champs d'une union peuvent être quelconques, y compris des structures ou des pointeurs.

C.3 Variables globales

Une variable globale est une variable qui est définie en dehors de toute fonction. Une variable globale déclarée au début d'un fichier source peut être utilisée dans toutes les fonctions du fichier. La variable n'existe qu'en un seul exemplaire et la modification de la variable globale dans une fonction change la valeur de cette variable dans les autres fonctions.

```
#include <stdio.h>

int x ;  /* déclaration en dehors de toute fonction */

void ModifieDonneeGlobale(void) /* pas de paramètre */
{
   x = x+1;
}

void AfficheDonneGlobale(void)  /* pas de paramètre */
{
   printf("%d\n", x);
}

int main(void)
{
   x = 1;
   ModifieDonneeGlobale();
   AfficheDonneGlobale();   /* affiche 2 */
   return 0;
}
```

Dans le cas d'un projet avec programmation multifichiers, on peut utiliser dans un fichier source une variable globale définie dans un autre fichier source en déclarant cette variable avec le mot clef `extern` (qui signifie que la variable globale est définie ailleurs).

```
extern int x;  /* déclaration d'une variable externe */
```

C.4 Do...while

La syntaxe `Do...while` permet d'itérer un bloc d'opérations comme la boucle while, mais en assurant que le bloc sera exécuter au moins une fois.

```
char SaisieYesNo(void)
{
   char reponse;
   do
     {
       puts("Répondez par oui (y) ou par non (n)");
       reponse = getchar();
       getchar();
     }
   while (reponse != 'y' && reponse != 'n');
   return reponse;
}
```

C.5 i++ ET ++i

Pour incrémenter une variable `i`, on peut utiliser `i++` ou `++i`. Dans la plupart des expressions, cela ne fait pas de différence. Cependant si la **valeur** de `i++` ou `++i` est utilisée, elle est différente : la valeur de `i++` est la valeur de `i` avant incrémentation, alors que la valeur de `++i` est la valeur de `i` après incrémentation.

Exemple

```
#include <stdio.h>

int main(void)
{
   int i, tab[3]={0,1,2};
   i = 1;
   printf("%d", tab[i++]);  /* affiche 1 */
   printf("%d", i);  /* affiche 2 */
   i = 1;
```

```
    printf("%d", tab[++i]);  /* affiche 2 */
    printf("%d", i);  /* affiche 2 */
    return 0;
}
```

C.6 Le générateur aléatoire : fonction rand

On tire un nombre aléatoire avec la fonction `rand` de la bibliothèsque `stdlib.h`, qui retourne un nombre aléatoire entre 0 et `RAND_MAX` (défini comme égal à 2147483647 dans `stdlib.h`). On peut faire appel à un modulo ou à un facteur d'échelle pour avoir un nombre dans une fourchette donnée.

Cependant, la fonction `rand` est implémentée par un algorithme déterministe, et si l'on souhaite obtenir toujours des nombres différentes, le générateur aléatoire doit être initialisé en fonction de l'heure. Pour cela, on utilise la fonction `srand`, qui permet d'initialiser le générateur aléatoire à une certaine valeur, et on peut utiliser la bibliothèque `time.h`, et par exemple la fonction `time`, qui retourne le nombre de secondes depuis le premier janvier 1970 à 0*h* (en temps universel).

Exemple

Le programme suivant affiche uns série de 10 nombres réels aléatoires entre 0 et 1 :

```
%4
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
  int i;
  double x;
  srand(time(NULL));
  for (i=0 ; i<10 ; i++)
    {
      x = rand()/((double)RAND_MAX);
      printf("%.4f\n", x);
    }
  return 0;
}
```

C.7 break ET continue

L'instruction `break` permet d'interrompre une boucle (`for`, `while`,`do...while`) et de passer à la suite.

```
int SaisieTableau(int *tab)
{
   int n=0, i;
   while (1)
      {
         printf("Entrez un élément : ");
         scanf("%d", &tab[n++]);
         printf("Voulez-vous continuer ? (y/n) ");
         if (getchar() == 'n')
            break;
         getchar();
      }
   puts("Vous avez tapé les nombres suivants : ");
   for (i=0 ; i<n ; i++)
      printf("%d ", tab[i]);
   puts("");
   return n;
}
```

L'instruction `continue` permet de passer directement à l'itération suivante dans une boucle.

```
#include <stdio.h>
#include <time.h>  /* pour utiliser time */
#include <stdlib.h>   /* pour utiliser rand et srand */

void GenereTableauAleatoire(int *tab)
{
   int n=0;
   while (n<99)
      {
         tab[n] = rand(); /* génération d'un nombre aléatoire */
         printf("Le nombre tiré est %d.")
         puts("Voulez-vous l'enregistrer ? (y/n)");
         if (getchar() == 'n')
            continue;
         n++;
         printf("Voulez-vous continuer ? (y/n) ");
         if (getchar() == 'n')
            break;
```

```
        getchar();
      }
   return n;
}

int main(void)
{
   int tab[100], n;
   srand(time(NULL)); /* initialisation générateur aléatoire */
   n = GenereTableauAleatoire(tab);
   Affiche(tab, n);
   return 0;
}
```

C.8 Macros

Le `#define` est une directive de précompilation. Lorsqu'on définit une constante avec un `#define`, toute occurrence de la constante dans le code source est remplacée littéralement par sa valeur avant la phase de compilation proprement dite. En fait, on peut définir par `#define` non seulement des constantes, mais des expressions composées. On appelle cela une *macro*. Une macro est en général plus rapide qu'une fonction à l'exécution, et on peut utiliser des macros pour optimiser. Cenpendant, les macros ont tendance à engendrer des bugs si l'on ne les manie pas avec précaution.

```
#include <stdio.h>

#define Carre(x)(x*x)   /* définition d'une macro */

int main()
{
   int a = 2;
   float b = 3.0;

   printf("a carré = %d", Carre(a));  /* affiche 4 */
   printf("b carré = %f", Carre(b)); /* affiche 9.0000 */
   printf("erreur = %d", Carre(a+1));  /* affiche a+1*a+1 = 2a+1 = 5
*/
   Carre(a++);
   printf("%d", a);  /* affiche 4 : a++ est effectué deux fois */
}
```

Il faut être extrêmement méfiant lorsqu'on passe des arguments composés (tel que `a+1` ou `a++`) dans une macro, car l'expression est recopiée littéralement dans la macro, ce qui crée parfois des suprises.

C.9 atoi, sprinf ET sscanf

Parfois, un nombre nous est donné sous forme de chaîne de caractères dont les caractères sont des chiffres. Dans ce cas, la fonction `atoi` permet de réaliser la conversion d'une chaîne vers un `int`.

```
#include <stdio.h>

int main()
{
   int a;
   char s[50];

   printf("Saisissez des chiffres : ");
   scanf("%s", s);  /* saisie d'une chaîne de caractères */
   a = atoi(s);  /* conversion en entier */
   printf("Vous avez saisi : %d\n", a);
   return 0;
}
```

Plus généralement, la fonction `sscanf` permet de lire des données formatées dans une chaîne de caractère (de même que `scanf` permet de lire des données formatées au clavier ou `fscanf` dans un fichier texte).

```
#include <stdio.h>

int main()
{
   float x;
   char s[50];

   printf("Saisissez des chiffres (avec un point au milieu) : ");
   scanf("%s", s);  /* saisie d'une chaîne de caractères */
   sscanf(s, "%f", &x);  /* lecture dans la chaîne */
   printf("Vous avez saisi : %f\n", x);
   return 0;
}
```

Inversement, la fonction `sprintf` permet d'écrire des données formatées dans une chaîne de caractères (de même que `printf` permet d'écrire dans la console ou `fprintf` dans un fichier texte).

```
#include <stdio.h>

void AfficheMessage(char *message)
{
   puts(message);
}

int main()
{
   float x;
   int a;

   printf("Saisissez un entier et un réel : ");
   scanf("%d %f", &a, &x);
   sprintf(s, "Vous avez tapé : a = %d x = %f", a, x);
   AfficheMessage(s);
   return 0;
}
```

C.10 Arguments d'un programme

La fonction `main` d'un programme peut prendre des arguments en ligne de commande. Par exemple, si un fichier `monprog.c` a permis de générer un exécutable `monprog` à la compilation,

```
$ gcc monprog.c -o monprog
```

on peut invoquer le programme `monprog` avec des arguments

```
$ ./monprog argument1 argment2 argument3
```

> ***Exemple***
>
> La commande `cp` du bash prend deux arguments :
>
> `$ cp nomfichier1 nomfichier2`

Pour récupérer les arguments dans le programme *C*, on utilise les paramètres `argc` et `argv` du `main`. L'entier `argc` donne le nombre d'arguments rentrés dans la ligne de commande **plus** 1, et le paramètre `argv` est un tableau de chaînes de caractères qui contient comme éléments :

- Le premier élément `argv[0]` est une chaîne qui contient le nom du fichier executable du programme ;
- Les éléments suivants `argv[1], argv[2]`, etc... sont des chaînes de caractères qui contiennent les arguments passés en ligne de commande.

Le prototype de la fonction `main` est donc :

```
int main(int argc, char**argv);
```

Exemple

Voici un programme longeurs, qui prend en argument des mots, et affiche la longueur de ces mots.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char**argv)
{
   int i;
   printf("Vous avez entré %d mots\n", argc-1);
   puts("Leurs longueurs sont :");
   for (i=1 ; i<argc ; i++)
      {
         printf("%s : %d\n", argv[i], strlen(argv[i]));
      }
   return 0;
}
```

Voici un exemple de trace :

```
$ gcc longueur.c -o longueur
$ ./longueur toto blabla
Vous avez entré 2 mots
Leurs longueurs sont :
toto : 4
blabla : 6
```

C.11 `fgetc` ET `fputc`

C.11.1 Lire caractère par caractère

La fonction `fgetc` permet de lire un caractère dans un fichier et est analogue à la fonction `getchar` qui permet de lire un caractère au clavier. La fonction `fgetc` prend en paramètre le pointeur de fichier et retourne le premier caractère lu dans le fichier pointé. Le pointeur de fichier passe automatiquement au caractère suivant. Lorsque la fin du fichier est atteinte, la fonction `fgetc` retourne le caractère `EOF` (le caractère `EOF` est défini par un `#define` dans `stdio.h` et signifie *End Of File*. La constante `EOF` vaut -1). On peut tester la valeur retournée par `fgetc` pour savoir si la fin du fichier est atteinte.

Exemple

Voici une fonction qui affiche dans la console le contenu d'un fichier "monfichier.txt".

```
char AfficheFichier(void)  /* retourne 1 si erreur 0 sinon */
{
  FILE *fp;      /* pointeur de fichier */
  char car;
  fp = fopen("monfichier.txt", "rt");
  if (fp == NULL)
    return 1;     /* erreur d'ouverture de fichier */
  /* on trouve une affectation dans la condition du while : */
  /* lecture jusqu'à la fin du fichier : */
  while ((car=fgetc(fp))!=EOF)
    printf("%c", car);  /* affichage du caractère à l'écran */
  fclose(fp);
  return 0;   /* pas d'erreur */
}

int main(void)
{
  if (AfficheFichier() != 0) |comle test affiche le fichier
     puts("Erreur, fichier inexistant ou droits insuffisants !");
   return 0;
}
```

C.11.2 Écrire caractère par caractère

Pour écrire des caractères les uns à la suite des autres dans un fichier, on peut utiliser la fonction fputc, qui est analogue à la fonction putchar qui affiche un caractère. La fonction fputc prend en premier paramètre un char et en deuxième paramètre un pointeur de fichier, et écrit le caractère dans le fichier.

Exemple

Voici une fonction qui recopie un fichier dans un autre fichier :

```
%20
char RecopieFichier(void)  /* retourne 1 si erreur 0 sinon */
{
  FILE *fpr, *fpw;      /* pointeurs de fichier */
  char car;
  fpr = fopen("monfichier.txt", "r");
  fpw = fopen("copiefichier.txt", "w");
  if (fpr == NULL || fpw == NULL)
```

```
    return 1;     /* erreur d'ouverture de fichier */
  /* lecture jusqu'à la fin du fichier : */
  while ((car=fgetc(fpr))!=EOF)
    fputc(car, fpw); /* ecriture de car dans le fichier fpw */
  fclose(fpr);
  fclose(fpw);
  return 0;   /* pas d'erreur */
}

int main(void)
{
  if (CopieFichier())
     puts("Erreur d'ouverture de fichier !");
}
```

C.12 Arithmétique de pointeurs

Un tableau, en tant que type de donnée, est un pointeur sur son premier élément. Les différents éléments du tableau occupent des position consécutives (qui se suivent) dans la mémoire. On dit que la mémoire d'un tableau est *contiguë*.

Lorsqu'un pointeur p pointe sur une case d'un tableau, le pointeur p+1 pointe sur la case suivante du tableau, le pointeur p+2 sur la case d'après, et ainsi de suite (voir la figure C.1). En particulier, si `tab` est un tableau, un pointeur sur l'élément `tab[i]` est donné par `tab+i`, et `tab[i]` est synonyme de `*(tab+i)`.

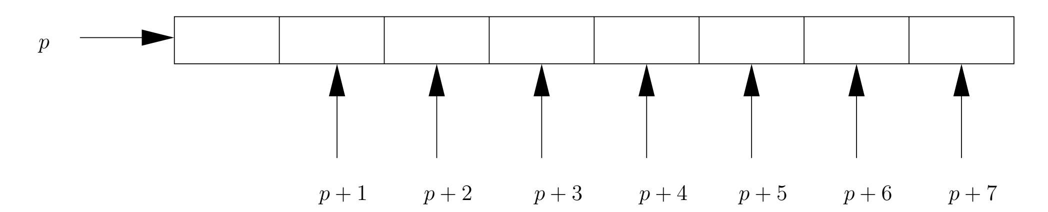


Figure C.1 – **L'arithmétique de pointeurs : un pointeur `p` et ses successeurs**

On peut aussi utiliser des pointeurs au lieu des indices pour parcourir les cases d'un tableau.

Exemple

La fonction de recherche suivante renvoie 1 si la chaîne de caractères passée en paramètre contient le caractère `c` passé en paramètre, et 0 sinon.

```
int Recherche(char *chaine, char c)
{
```

```
    char *p;
    p = chaine;   /* p pointe sur la première case de chaine */
    /* tant que la fin de chaine n'est pas atteinte : */
    while (*p != '\0')
      {
        if (*p == c)  /* test sur l'objet pointé par p */
          return 1;
        p = p+1;   /* passage à la case suivante */
      }
    return 0;
}
```

Exercices

fgetc et fputc

C.1 (∗) Écrire un programme qui calcule le nombre d'occurences de la lettre 'M' dans un fichier.

C.2 (∗) Écrire un programme qui recopie un fichier en ne mettant que les caractères alphabétiques. On pourra utiliser la fonction `isalpha`, qui prend en paramètre un caractère, et retourbe une valeur non nulle s'il s'agit d'un caractère alphabétique.

C.3 (∗∗∗) Écrire un programme qui compte les mots d'un fichier `"monfichier.txt"`. Un mot est une suite de caractères alphabétiques.

Arithmétique de pointeurs

Dans les exercices suivants, on n'utilisera aucun indice de tableau.

C.4 (∗) Faire une implémentation de la foction `strcpy`.

C.5 (∗) Faire une implémentation de la fonction `strcmp`.

C.6 (∗) Faire une implémentation de la fonction `strlen`.

C.7 (∗) Faire une implémentation de la fonction `strcat`.

C.8 (∗) Refaire la fonction de recherche ci-dessus pour qu'elle renvoie l'adresse de la première occurence du caractère `c` dans la chaîne.

C.9 (∗∗)

a) Faire une fonction qui prend en paramètre un tableau d'entiers et son nombre d'éléments et qui renvoie l'adresse de son élément maximum.

b) Faire une fonction qui prend en paramètre un tableau d'entiers et son nombre d'éléments et qui renvoie l'adresse de son élément minimum.

c) En utilisant les fonctions du a) et du b), faire une fonction qui échange le minimum et le maximum d'un tableau.

Corrigés

C.1

```
int main (void)
{
  FILE *fp;
  int compteur=0;
  char c;
  fp =fopen("Fichier.txt", "r");
  if (fp == NULL)
    {
      printf("Erreur d'ouverture du fichier\n");
      exit (1);
    }
  while ( (c =fgetc(fp))  != EOF )
    if ( c == 'M')
      compteur++;
  printf("Le nombre d'occurence de la lettre M est %d\n",compteur);
  return 0;
}
```

C.2

```
#include <ctype.h>

int main (void)
{
  FILE *fp,*fo;
  char c;
  fp =fopen("Fichierin.txt", "r");
  fo =fopen("Fichierout.txt", "w");
  if (fp == NULL || fo == NULL)
```

```
    {
      printf("Erreur d'ouverture du Fichierin ou Fichierout\n");
      exit (1);
    }
  while ( (c =fgetc(fp))  != EOF )
    if ( isalpha(c))
      fputc(c,fo);
  return 0;
}
```

C.3

```
#include <ctype.h>

int main (void)
{
  FILE *fp;
  int compteur=0,flagcompteur=0,flag=0;
  char c;
  fp =fopen("Fichier.txt", "r");
  if (fp == NULL)
    {
      printf("Erreur d'ouverture du fichier \n");
      exit (1);
    }
  while ( (c =fgetc(fp))  != EOF )
    {
      if (isalpha(c)) /*si le caractère lu est alphabétique*/
        flagcompteur=1;
      else
        if ((c==' ' || c=='\n')) /* si le caractère lu est un espace */
                                 /* ou un retour à la ligne */
          {
            if (flagcompteur==1)
              {
                compteur ++;
                flagcompteur=0;
              }
          }
        else /*si le caractère lu n'est pas alphabétique,
       /* ni un espace ni un retour à la ligne */
          {
            flagcompteur=0;
            flag=0;
            while(flag==0)
```

```
                {
                  c =fgetc(fp);
                  if (c==' ' || c=='\n' || c ==EOF)
                    flag=1;
                }
            }
      }
    if (c=EOF && flagcompteur==1)
      compteur ++;
    printf("Le nombre de mots du fichier est : %d",compteur);
    return 0;
}
```

C.4

```
char * strcpy(char *dest,char *src)
{
    int i=0;
    while(*(src+i)!='\0')
      {
        *(dest+i)=*(src+i);
        i++;
      }
    *(dest+i)=*(src+i); /* copier le '\0' */
    return dest;
}
```

C.5

```
int strcmp(char *s1,char *s2)
{
    char uc1, uc2;
    while (*s1 != '\0' && *s1 == *s2)
      {
        s1++;
        s2++;
      }
    uc1 = *s1;
    uc2 = *s2;
    return ((uc1 < uc2) ? -1 : (uc1 > uc2));
}
```

C.6

```
size_t strlen( char *str)
{
    char *p;
```

```
  size_t taille=0;
  p=str;/*p pointe vers la première case de str*/
  while(*p!='\0')
    {
      taille++;
      p=p+1;
    }
  return taille;
}
```

C.7

Dans cet exercice, de même que dans la bibliothèque `stdlib.h`, on suppose que `str1` est alloué suffisemment long pour contenir la concaténation de `str1` et `str2`.

```
char *strcat(char *str1,char *str2)
{
  char *p1,*p2;
  p1=str1;/* p1 pointe vers la première case de str1 */
  p2=str2;/* p2 pointe vers la première case de str2 */
  while(*p1!='\0') /* parcourir la chaine str1 */
    p1=p1+1;

  while(*p2!='\0') /* parcourir la chaine str2 */
    {
      *p1=*p2;
      p1++,p2++;
    }
  *p1='\0'; /*rajouter le '\0' à la fin de la concaténation*/
  return str1;
}
```

C.8

```
char* RechercheModifie(char *chaine,char c)
{
  char *p;
  p=chaine;
  while(*p!='\0')
    {
      if (*p==c)
        return p;
      p=p+1;
    }
  return NULL;
}
```

C.9

a)

```
int * Maximum(int *tab,int n)
{
  int max=*tab,i;
  int *addr=tab;
  for (i=1;i<n;i++)
    if (*(tab+i)>max)
      {
        max=*(tab+i);
        addr=tab+i;
      }
  return addr;
}
```

b)

```
int * Minimum(int *tab,int n)
{
  int min=*tab,i;
  int *addr=tab;
  for (i=1;i<n;i++)
    if (*(tab+i)<min)
      {
        min=*(tab+i);
        addr=tab+i;
      }
  return addr;
}
```

c)

```
void Echangeminmax(int *tab,int n)
{
  int tmp;
  int *addr1,*addr2;
  addr1=Minimum(tab,n);
  addr2=Maximum(tab,n);
  tmp=*addr1;
  *addr1=*addr2;
  *addr2=tmp;
}
```

Index

Symbols

A

B

C

D

E

F


DUNOD